UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Graph Neural Networks and its applications

Master in Innovation and Research in Informatics

Master's Thesis

Author:
Pau Rodríguez Esmerats

Thesis supervisor:
Prof. Marta Arias Vicente

October 13, 2019

Contents

To my beloved Bruna and Alba, for making my world a happy place.

Abstract

Data structured as graphs exists in multiple domains like biochemistry, image processing, recommender systems and social network analysis to name a few. Several approaches to train machine learning models on graph structured data by using preprocessing techniques exist but they lack the flexibility to completely adapt to the dataset and task at hand. Graph Neural Networks allow to create an end to end machine learning model that is simultaneously trained to learn a representation of graph structured data and to fit a predictive model on it. Graph Neural networks can be applied to tasks that range from clustering or visualization, to classification or regression on graph data, and they can also learn a representation at the node or graph level. They achieved state-of-the-art performance on semi-supervised learning and supervised graph classification. Semi-supervised learning is when a graph contains some labeled nodes and their labels are extended to the rest of the nodes of the graph, for example to classify Reddit posts. Supervised graph classification or regression is applied for predicting the class or the value of some attributes of a whole graph or a subgraph. This work will explore some of the most prominent Graph Neural Network variants and apply them to two tasks: approximation of the community detection Girvan-Newman algorithm and compiled code snippet classification.

1 Introduction

Data structured as graphs exists in many domains like biology, chemistry, image processing, recommender systems and social networks analysis to name a few. Using this data in a machine learning model has proven to be difficult due to the high dimensionality and non-Euclidean properties of graph data. Over the years, several approaches to train machine learning models on graph structured data by summarizing or representing the information in a simplified way have been used. However, those approaches are used as a preprocessing step, not being part of the training process. Graph Neural Networks, a recent novel technique, allows to create an end to end machine learning model that is simultaneously trained to learn a representation of graph structured data and to fit a predictive model on it.

Graph Neural Networks [1] learn a representation of the graph in a way that it creates an embedding of the graph nodes in a low dimensional space. The end to end training allows this representation to be learned with the purpose of reflecting the structural properties of the graph that are of interest for the problem at hand. The representation of nodes as an embedding is created in an iterative way by aggregating information from the neighbors of each node. This process is computationally costly, so modern implementations improve the speed by limiting the number of iterations or by using sampling techniques. This node embedding representation can be used by a downstream machine learning that can also be trained at the same time as the embedding. When the task requires to classify or to do a regression at the node level, the representation of each node in the embedding is directly used. When the task requires to classify or predict values at the graph or subgraph level, then pooling techniques can be used to obtain a graph or subgraph level representation.

The most prominent improved versions of Graph Neural Networks apply the ideas of Convolutional Neural Networks to graphs, therefore known as Graph Convolutional Networks [2]. They can be categorized in two main families, the spectral based methods and the spatial based methods. Spectral-based methods rely on the eigen-decomposition of the adjacency matrix of the graph, and this makes them less suitable for processing large data or for generalizing to unseen data. On the other hand, spatial-based methods rely on the aggregation of information from the neighborhood of each node, which allow the algorithm to process the graph in batches and so to be able to process large graphs.

State-of-the-art performance has been attained for semi-supervised learning on large graphs, where a small proportion of nodes have a label or target value and the rest are unlabeled. The task consists in assigning labels to the rest of nodes that are unlabeled ([2], [3]). There has been also a breakthrough in the task of graph classification and regression in protein analysis [4]. That is one of the reasons that the scientific community is getting more interest in that area, with a significant increase of publications since 2016.

In those recent years, publications showing the application of Graph Neural Networks have appeared in the domains of biochemistry, computer vision, recommender systems, combinatorial optimization, traffic optimization, inductive logic and program verification. The main tasks that Graph Neural Networks solve can be summarized as node(graph) classification, node(graph) regression, link prediction, node clustering, graph partition and graph visualization.

The goal of this master's thesis is to apply Graph Neural Network models to different problems to create a novel solution. The idea is to get to know how Graph Neural Networks are used in each situation. Two problems are explored: Girvan-Newmann algorithm approximation and compiled code function classification. They correspond to the two main tasks that Graph Neural Network perform with success: semi-supervised learning of nodes on a graph and supervised graph classification.

Graph Neural Networks seem to be a promising way of solving graph-related problems, with applications in many domains. The time seems right to jump into learning about the most recent models since they have attained state-of-the-art performance on some of the tasks they have solved.

The thesis is organized in the following way: the next section will explain the state-of-the-art in Graph Neural Network models by presenting the main models, their internals and the problems they have solved. Then, in section three, the methodology followed in the experiments is presented. After that, section four will go through an overview of the implementation of the experiments, whereas in section five the results of the experiments are summarized. Finally, in section six the conclusion of the thesis is presented.

2 State of the art

This section is a compilation of several sources that survey the most recent Graph Neural Network advances as well as giving detailed information on their internals. Parts of the text are extracted from the research papers and surveys that follow:

- Hamilton et al., Representation Learning on Graphs: Methods and Applications [5]
- Xu et al., How Powerful are Graph Neural Networks? [6]
- Wu et al., A comprehensive Survey on Graph Neural Networks [7]
- Kipf et al., Semi-supervised classification with Graph Convolutional Networks [2]
- Li et al., Gated graph sequence neural networks [8]

Many data can be represented as a graph, a data structure employed often in fields like biochemistry, social networks, recommender systems and even computer program analysis. Many machine learning applications are created to make predictions using graph-structured data. The way to incorporate the information from graph-structured data as an input to a machine learning model is not straightforward. The fact that graph-structure data is high-dimensional and non-Euclidean are the main barriers for creating a unified and general way to use it as an input to a model. Graph data is irregular, with variable size and variable number of neighbors. Many approaches to transform graph data into usable features for machine learning models exist by using summary graph statistics, kernel functions, or hand-engineered features. Those approaches lack the flexibility to adapt during the learning process.

The idea behind graph neural networks is to learn a mapping that embeds nodes, or entire subgraphs, as points in a low-dimensional vector space R^d. The goal is to optimize this mapping so that geometric relationships in the embedding space reflect the structure of the original graph. Previous approaches to learn on graphs used this approach as a preprocessing step, with fixed and/or hand engineered parameters. The graph neural networks treat this representation learning task as a machine learning task itself, using a data-driven approach to learn the embeddings that encode the desired graph structure.

This section will present the main advances in graph neural networks, showing the techniques for representing nodes and entire subgraphs.

2.1 Notation

Definition of a Graph: A graph is defined as $G = (V, E, A)$ where V is the set of nodes, E is the set of edges and A is the adjacency matrix. In a graph, let $v_i \in V$ denote a node and $e_{i,j} = (vi, vj) \in E$ denote an edge. The adjacency matrix A is a N x N matrix with $N = |V|$ where $A_{i,j} = w_{i,j} > 0$ if $e_{i,j} \in E$ and $A_{i,j} = 0$ if $e_{i,j} \notin E$. The degree of a node is the number of edges connected to it, formally defined as $degree(v_i) = \sum A_{i,:}$

A graph can be associated with node attributes X, where $X \in R^{N \times D}$ is a feature matrix with $X_i \in R^D$ representing the feature vector of node v_i.

Definition of a directed graph: A directed graph is a graph with all edges pointing from one node to another. For a directed graph, $A_{i,j} \neq A_{j,i}$. An undirected graph is a graph where all edges are bidirectional. For an undirected graph, $A_{i,j} = A_{j,i}$.

Euclidean space: a space in any finite number of dimensions, in which points are designated by coordinates (one for each dimension) and the distance between two points is given by a distance formula. A distance d is defined as a function, let $x, y \in \mathbb{R}^p$:

$$d : \mathbb{R}^{2p} \to \mathbb{R}_+ \begin{cases} d(x,y) > 0 & \forall x \neq y \\ d(x,y) = 0 & iff \quad x = y \\ d(x,y) \leq d(x,z) + d(z,y) & \forall x, y, z \end{cases}$$

An Euclidean distance d between two points x and y is defined as $d^2(x,y) = (x-y)^T A(x-y)$ where $\mathcal{A}$ is a positive definite matrix, called a metric.

Graph-structured data is considered to be non Euclidean. There's not a clear definition of the norm of a vector representing a node in the space defined by the graph itself. Consequently, the distance between two nodes has to be defined on some other criteria. Usually the distance between two nodes is computed as the number of nodes that exist in the shortest path following edges between the node at the origin and the node at the destination. Moreover, the similarity between nodes, as computed based on the node's attributes, does not need to comply with node distances (as defined by previous sentence).

2.2 Original model

The idea of a graph neural network was first presented by Gori et al.(2005) [9] and then developed by Scarselli et al. (2009) [1]. Good and generic definitions of the process of training a Graph Neural Network are explained at [5] and [6]. They are summarized in the following paragraphs.

Graph Neural Networks: GNNs use the graph structure and node features X_v for $v \in V$ to learn a representation vector of a node, h_v, or the entire graph h_G. Modern GNNs follow a neighborhood aggregation strategy, where they iteratively update the representation of a node by aggregating representations of its neighbors. After k iterations of aggregation, a node's representation captures the structural information within it's k-hop network neighborhood. Formally, the k-th layer of a GNN is:

$$a_v^{(k)} = AGGREGATE^{(k)}(\{h_u^{(k-1)} : u \in N(u)\}),$$

$$h_v^{(k)} = COMBINE^{(k)}(h_v^{(k-1)}, a_v^{(k)}),$$

where $h_v(k)$ is the feature vector of node v at the k-th iteration/layer. The initialization consists in $h_v^{(0)} = X_v$, and $N(v)$ is a set of nodes adjacent to v. The iterative aggregation process is computationally intensive. Modern implementations try to speed up this convergence process, for example by limiting the number of iterations or by using random walks for processing the neighborhood. To ensure convergence, the recurrent function (the composition of $COMBINE$ and $AGGREGATE$) must be a contraction mapping. A contraction map shrinks the distance between two points after mapping. GNN use the Almeida-Pineda algorithm [10] to train the model. The core idea is to run the propagation process to reach a convergence point and then perform the backward procedure given the converged solution.

There are two types of tasks, where GNNs are used: node classification/regression and graph classification/regression. Node classification or regression is when each node $v \in V$ has an associated label or target value y_v and the goal is to learn a representation vector h_v of v such that v's label or target values can be predicted as $y_v = f(h_v)$. Graph classification or regression is when given a set of graphs $\{G1, ..., G_N\}$ and their labels or target values $\{y_1, ..., y_N\}$ the goal is to learn a representation vector h_G that helps to predict the label or target value of the entire graph $y_G = g(h_G)$.

For node classification/regression, the node representation $h_v^{(K)}$ of the final iteration is used for prediction. For graph classification/regression, the READOUT function aggregates node features from the final iteration to obtain the entire graph's representation h_G:

$$h_G = READOUT(\{h_v^{(K)} | v \in G\})$$

The READOUT function can be a simple permutation invariant function such as a summation or a more sophisticated graph-level pooling function.

The choice of $AGGREGATE^{(k)}(\cdot)$ and $COMBINE^{(k)}(\cdot)$ is crucial. In the next subsection the main evolutions of this model is explained. The main idea is what choices of $AGGREGATE^{(k)}(\cdot)$ and $COMBINE^{(k)}(\cdot)$ have been used for those state-of-the-art models.

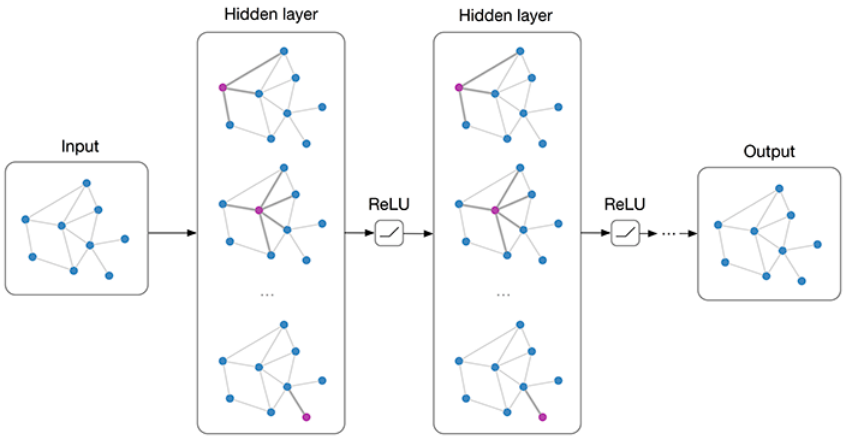

Figure 1: Graph Neural Network architecture (from Thomas Kipf et al. 2016 [2])

The figure 1 shows an example of a 2 layer Graph Neural Network embedding. In each "Hidden layer" of the figure, the model is performing the aggregation and the combination (with readout if needed) phase before producing its output.

2.3 Advanced models

Early models try to learn a node's representation by propagating neighbor information in an iterative manner until convergence in all nodes is reached. This process is computationally intensive and so recently there have been several publications that try to make this process less costly. A large number of methods apply the characteristics of convolutional networks in the computer vision domain to graph data. They are called Graph Convolutional Networks (GCN).

Graph Convolutional network:

The first GCN model is presented in Bruna et al.(2013) [11] which develops a variant of graph convolution based on spectral graph theory. There have been several other publications on spectral-based graph convolutional networks [12]. Spectral methods define graph convolutions by introducing filters from the perspective of graph signal processing, where the graph convolution operation is interpreted as removing noise from graph signals. They rely on the eigen-decomposition of the Laplacian matrix. This has the drawback that any perturbation of a graph results in a change of eigen basis, and the learned filters are domain dependent so they can't be applied to a graph with a different structure.

There is another kind of convolutional approach for graphs, called spatial-based graph convolutional networks, see [3] and [13]. They define graph convolution based on a node's spatial relations. These methods directly perform the convolution in the graph domain by aggregating the neighbor nodes information. The spatial-based graph convolution takes the aggregation of a node and its neighbors to get a new representation for it. A common practice is to stack multiple graph convolution layers together.

In terms of scalablity and parallelization, the spectral methods cost increments exponentially with the size of the graph but it needs the whole graph in memory. So these methods are not suitable for large scale data or parallelized architectures. However, spatial methods perform the convolution in the graph domain via aggregating the neighbors attributes so they can handle large graphs. The computation can be performed on batches of nodes instead of the whole graph. Sampling techniques can also be used when the number of neighbors becomes prohibitive.

In terms of generalization of the models, the spectral-based models assume a fixed graph and are not good at generalizing to unseen graphs. Spatial-methods don't have this constraint as their convolution is performed locally and the weights used in the convolution can be shared across different locations.

Finally, spectral methods are limited to work on undirected graphs whereas spatial methods can deal with multi-source inputs and directed graphs by modifying the aggregation function. Spatial-based models are attracting more attention on the scientific comunity than spectral-based models.

Let's see how the spatial-based graph convolutional network implements the algorithm that will iteratively generate a representation of the nodes. This algorithm depends on weights, also called coefficients, that will be later updated with stochastic gradient descent or equivalent algorithms. These updating mechanism allows for training an end to end model that contains the representation of the graph generation(the embedding) as well as the subsequent layers that will use the embedding as an input for their classification or regression task.

The previous subsection showed that the updating of a node state (value of its attributes), can be formalized as a function of the node states and the aggregation of the neighbor nodes states:

$$h_v^{(k)} = COMBINE^{(k)}(h_v^{(k-1)}, a_v^{(k)})$$

,

where $a_v^{(k)}$ is the aggregation of the neighborhood. This same concept can be expressed in matrix notation with the adjacency matrix and matrix of weights in the following equation:

$$H^{(k)} = f(H^{(k-1)}, A) = \sigma(AH^{(k-1)}W^{(k-1)}),$$

where $W^{(k)}$ is a weight matrix for the l-th neural network layer and $\sigma(\Delta)$ is a non-linear activation function like ReLU. Some additional constraints must be imposed on the adjacency matrix A. First, it must be added to the identity I, in order to create self-loops and allow the node attributes to be counted in the updating. Second, A must be normalized.

This functions is implementing the Weisfeiler-Lehman algorithm on graphs, which is used to detect isomorphisms. It assigns a unique set of features for each node that uniquely describes its role in the graph. It works as follows:

For all nodes $v_i \in G$:

- Get features $\{h_{vj}\}$ of neighboring nodes $\{v_j\}$
- Update node feature $h_{v_i} = hash(\sum_j h_{vj})$, where $hash(\cdot)$ is an injective hash function.
- Repeat for k steps until convergence.

Variations of this idea conform the foundations of the more advanced Graph Neural Network models seen in this subsection, the Graph Convolutional Network and its variants, like GraphSage or the Graph Isomorphism Network called GIN.

The Graph Convolutional Network model uses filter parameters $W^{(k)}$ that are shared over all locations in the graph:

$$h_{v_i}^{(k)} = \sigma(\sum_j \frac{1}{c_{ij}} h_{v_j}^{(k-1)} W^{(k-1)})$$

where j indexes the neighboring nodes of v_i, and c_{ij} is a normalization constant for the edge (v_i, v_j). This version is a differentiable and parameterized variant of the hash function used in the original Weisfeiler-Legman algorithm. With proper initialization (Glorot) this update rule is stable.

When the model is trained to perform classification on the whole graph or a subgraph, a graph pooling module must be used. GCNs that use pooling modules are as powerful as the Weisfeiler-Legman test in distinguishing graph structures. Similar with the pooling layer of CNNs, graph pooling module could easily reduce the variance and computation complexity by down-sampling in between convolutional layers. Mean, max and sum pooling are the most common way to implement them.

The remaining part of the subsection will present evolutions of the GCN model that are most commonly known.

Message Passing Neural Networks (MPNN): The MPNN model generalizes several existing graph convolution networks into a unified framework named Message Passing Neural Network. It consists of two phases, the message passing phase and the readout phase. The message passing phase actually runs T-step spatial-based graph convolutions. The graph convolution operation is defined through a message function $M_t(\cdot)$ and an updating function $U_t(\cdot)$ according to

$$h_v^t = U_t(h_v^{(t-1)}, \sum_{w \in N(v)} M_t(h_v^{t-1}, h_w^{t-1}, e_{vw}))$$

The readout phase is actually a pooling operation which produces a representation of the entire graph based on hidden representations of each individual node. It is defined as

$$y' = R(h_v^T | v \in G)$$

Through the output function $R(\cdot)$, the final representation y' is used to perform graph-level prediction tasks. The authors [14] state that several other graph convolution networks fall into their framework by assuming different forms of $U_t(\cdot)$ and $M_t(\Delta)$.

GraphSage: It introduces in [3] the notion of the aggregation function to define the graph convolution. The aggregation function essentially assembles a node's neighborhood information. It must be invariant to permutations of node orderings such as mean, sum and max. The graph convolution operation is defined as,

$$h_v^t = \sigma(W^t \cdot aggregate_k(h_v^{t-1}, \{h_u^{k-1}, \forall u \in N(v)\}))$$

Instead of updating states over all nodes, GraphSave proposed a batch-training algorithm, which improves scalability for large graphs. The learning process of GraphSage consists of three steps. First, it samples a node's local k-hop neighborhood with fixed-size. Second, it derives the central node's final state by aggregating its neighbors feature information. Finally, it uses the central node's final state to make predictions and back-propagate errors. Assuming the number of neighbors to be sampled at the t^{th} hop is s_t, the time complexity of GraphSage in one batch is $O(\prod_{t=1}^{T} s_t)$. Therefore the computation cost increases exponentially with the increase of t. This prevents GraphSage from having a deep architecture. However, in practice, the authors find that with t=2 GraphSage already achieves high performance.

Gated Graph Neural Networks (GGNN): also known as GGNN, this kind of GNN employ the gated recurrent units (GRU) [15] as the recurrent function, reducing the recurrence to a fixed number of steps. The spatial graph convolution of GGNNs is defined as

$$h_v^t = GRU(h_v^{(t-1)}, \sum_{u \in N(v)} W h_u^t)$$

For updating of the weights, the GGNNs use back-propagation through time (BPTT) to learn the parameters. The advantage is that it no longer needs to constrain parameters to ensure convergence. However, the downside of training by BPTT is that it sacrifices efficiency both in time and memory. This is especially problematic for large graphs, as GGNNs need to run the recurrent function multiple times over all nodes, requiring intermediate states of all nodes to be stored in memory.

Other types of GGNs are the Graph attention Networks, Graph Auto-encoders, Graph Generative Networks and Graph Spatio-temporal Networks.

Graph Attention Networks are similar to GCNs and seek an aggregation function to fuse the neighboring nodes, random walks, and candidate models in graphs to learn a new representation. The key difference is that graph attention networks employ attention mechanisms which assign larger weights to the more important nodes, walks, or models. The attention weight is learned together with neural network parameters within an end-to-end framework.

Graph Auto-encoders are unsupervised learning frameworks which aim to learn a low dimensional node vectors via an encoder, and then reconstruct the graph data via a decoder. Graph autoencoders are a popular approach to learn the graph embedding, for both plain graphs without attributed information as well as attributed graphs. For plain graphs, many algorithms directly pre-process the adjacency matrix, by either

constructing a new matrix (i.e., point-wise mutual information matrix) with rich information, or feeding the adjacency matrix to an autoencoder model and capturing both first order and second order information. For attributed graphs, graph autoencoder models tend to employ GCN as a building block for the encoder and reconstruct the structure information via a link prediction decoder.

Graph Generative Networks aim to generate plausible structures from data. Generating graphs given a graph empirical distribution is fundamentally challenging, mainly because graphs are complex data structures. To address this problem, researchers have explored to factor the generation process as forming nodes and edges alternatively, to employ generative adversarial training. One promising application domain of graph generative networks is chemical compound synthesis. In a chemical graph, atoms are treated as nodes and chemical bonds are treated as edges. The task is to discover new synthesizable molecules which possess certain chemical and physical properties.

Graph Spatial-temporal Networks aim to learn unseen patterns from spatial-temporal graphs, which are increasingly important in many applications such as traffic forecasting and human activity prediction. For instance, the underlying road traffic network is a natural graph where each key location is a node whose traffic data is continuously monitored. By developing effective graph spatial temporal network models, we can accurately predict the traffic status over the whole traffic system. The key idea of graph spatial-temporal networks is to consider spatial dependency and temporal dependency at the same time. Many current approaches apply GCNs to capture the dependency together with some RNN or CNN to model the temporal dependency.

2.4 Alternatives to Graph Neural Networks

Alternatives to Graph Neural Networks existed long before their first appearance: summary graph statistics, kernel functions and network embeddings (matrix factorization and random walks).

Summary statistics aim to capture very specific features of the whole graph, hoping that they will allow the model to discriminate or predict values. The most common ones are the average node degree, the clustering coefficient as well as the assortativity. They usually lack a lot of representation flexibility and fail to correctly capture aspects of the graph structure that are not reflected by those summary statistics

Kernel functions allow to capture much more signal from the graph structure, and then can be used to build powerful classification or regression algorithms. They are also used in unsupervised learning approaches. The drawback from kernels is that they are not suitable for large scale graphs, as they need to have the whole graph in memory.

Network embeddings aim to represent network vertices's into a low-dimensional vector space, by preserving both network topology structure and node content information. This embedding can be used in any downstream shallow machine learning algorithm, such as classification, regression, or clustering, in the same way that the output of a GGN is. Many network embedding algorithms are typically unsupervised algorithms and they can be broadly classified into three groups: matrix factorization [16], random walks [17], and deep learning approaches.

2.5 Applications of Graph neural Networks

Finally, this section presents the most important applications of Graph Neural Networks and its variants. The applications in several specific domains all fall under the following list of machine learning tasks to which GNNs are showing good results:

- node classification/regression
- graph classification/regression
- graph generation
- graph visualization
- node clustering

- link prediction
- graph partition

Computer Vision: in this domain, research has been done to perform scene graph generation, point cloud classification and segmentation, and action recognition. In scene graph generation, semantic relationships between objects facilitate the understanding of the semantic meaning behind a visual scene. Given an image, scene graph generation models detect and recognize objects and predict semantic relationships between pairs of objects [18]. In point clouds classification and segmentation, [19], the task is to identify the objects depicted by point clouds, by using graph convolution networks to explore the topological structure. In action recognition, persons recognized in an image(or video) have their joints detected and their connections form a graph that can be analyzed with spatial-temporal neural networks to classify human actions.

Recommender Systems: graph based recommender systems take items and users as nodes. By leveraging the relations between items and items, users and users, users and items, as well as content information, graph-based recommender systems are able to produce high-quality recommendations. The key to a recommender system is to score the importance of an item to an user. As a result, it can be cast as a link prediction problem. The goal is to predict the missing links between users and items([20],[21],[22]).

Biochemistry: graph neural networks are used to classify the molecules structure. Nodes represent atoms and edges represent chemical bonds. Node classification, graph classification and graph generation are the three main tasks. Examples include learning molecular fingerprints [23], predict molecular properties [14] and to synthesize chemical compounds [24]

Communications Network modeling: Communication networks rely on routing and path optimization to ensure good level of service. Nowadays network operators lack efficient network models able to make accurate predictions of relevant end-to-end Key Performance Indicators (KPI) such as delay or jitter at limited cost. Analytic models and packet-level simulations have been used to produce KPI's but at a high cost. A graph neural network model, RouteNet [25], has been proposed as an efficient solution to produce estimations of delay and jitter with similar accuracy as packet-level network simulators.

Program verification: is another fruitful domain where Graph Neural Networks are showing their potential ([8], [26] and [27]). Instructions follow a sequence of execution with loops, branches and calls to other instructions. This kind of modeling of the code allows for verification of correct variable usage (avoid usage after free of a memory buffer), variable name guessing and function renaming for correct understanding of code functionality. Gated Graph Sequence Neural Network [8], known as GGNN, are Graph Neural Networks that make use of gated recurrent units and modern optimization techniques that can output sequences. They poses a favorable inductive bias relative to purely sequence-based models like LSTMs, when the problem is graph-structured. They achieve state-of-the-art performance on program verification applied to memory safety, to map the state of the memory as a set of input graphs into a logical description of the data structures. For example this can be used to verify that there are no null pointer dereferences in a program. In [26], the authors use graphs to represent both the syntactic and semantic structure of code and use graph-based deep learning methods to learn to reason over program structures. They apply a modified version of GGNN to the task of VarNaming (predicting the name of a variable given its usage) and VarMisuse (reason about selecting the correct variable in a given program location). Finally, an approach to predicting names for entire functions or subroutines is shown in [27], where no Graph Neural Network approach has been used, instead an embedding is generated from Abstract syntax trees. It is used for snippets of code to predict semantic properties. We believe that Graph Neural Networks have great potential to achieve state of the art performance.

Social Network analysis: With the rapid expansion of the web, the last years have witnessed large adoption of social networks as one of the main ways to interact with people. Graph Neural Networks have many direct applications in social network analysis as friend or link prediction [28], community detection [2] and many more as can be seen in [5] and [29]. One of the main tasks in social network analysis is the process of discovering communities, known as community detection. The Girvan-Newman algorithm [30] is a clustering based approach to perform community detection, and is considered one of the most powerful algorithms. It proposed a divisive algorithm based on edge-betweenness for a graph with undirected and unweighted edges. The algorithm focused on edges that are most "between" the communities and communities are constructed progressively by removing these edges from the original graph.The worst-case for time complexity of the edge-betweenness algorithm is $O(m^2n)$ and is $O(n^3)$ for sparse graphs, with m being the number of edges and n the number of vertices's. Its downside is that it is very time consuming which makes it not suitable for applying it to modern social networks. As surveyed by [31], many authors have worked to improve it, reduce the computational complexity or extract useful concepts from it (see [32] and [33], [34], [35]).

3 Methodology

The methodology used in this thesis is based on the CRISP-DM methodology, as well as the well known hyper-parameter search performed by cross-validation. Any other insights that are extracted from the data follow the standards of design of experiments and statistical hypothesis modeling.

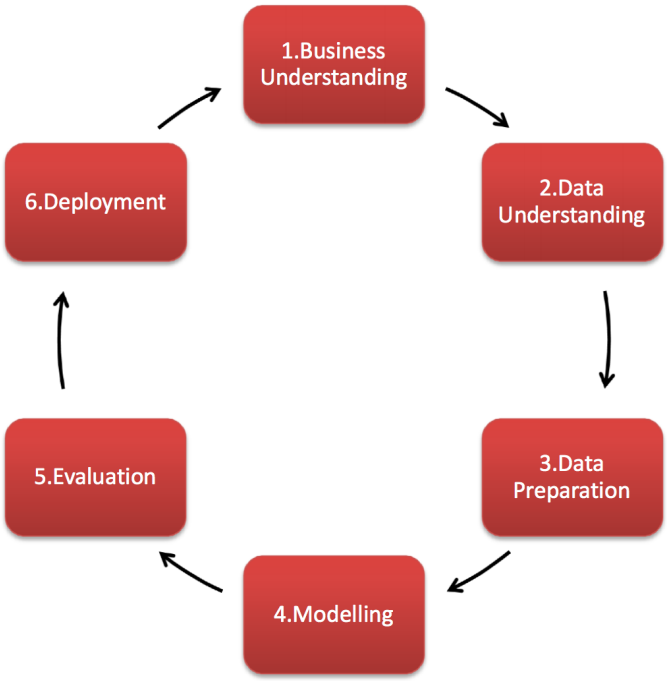

Figure 2: CRISP-DM methodology steps

CRISP-DM is the most representative method to plan the overall data extraction or generation, design of experiments and evaluation. It consists of six major tasks: business understanding, data understanding, data preparation, modeling, evaluation and deployment. Since this thesis has exploratory and research goals, the first and last steps of the CRISP-DM, business understanding and deployment, are not implemented, simple indications are given about them. The following points give an overview of how the CRISP-DM has been implemented in each of the two experiments.

Problem understanding

This steps allows for careful review of the goals and motivations of the thesis and its experiments. It's the place where the experiments to be performed are designed and the desired goals are stated. It gives an indication on how to perform the experiments, with which data, and when to stop.

Data understanding

In this step, the selected data is analyzed to decide which preprocessing actions will be needed, which machine learning tasks are more suitable for it and which results can we expect from it. Some insights of the data are performed as a data exploratory analysis. Distributions of features, correlation analysis, generation of aggregations and other basic statistical analysis are performed when necessary. Specially for the second task of this project for which data is not labeled, data labeling procedures are analyzed and tested at this stage.

Data preparation

In this step, all the needed preprocessing is performed. It consists of the extraction of the actual data, and the execution of data cleaning, where outliers, errors and missing data are corrected in the most suitable way for the problem that needs to be solved. Then the feature selection is performed, where data features are chosen based on the goals and previous analysis. Feature engineering is also performed to come up with transformations of the data that are suitable for the problem at hand.

Modelling

At this stage, data is prepared to be fed to a model training procedure. The model training procedure will consist on performing a search for the best hyper-parameters for each type of model that is intended to be trained. Out of this hyper-parameter search, the best model of each type for the current dataset will be produced as an output. The hyper-parameter search is executed by a well-known procedure called cross-validation, where the dataset is split into training, validation and testing. The model is trained on the training split and tested on the validation split until the best set of parameters is found. Then finally, to assess the generalization power of the best model, it is tested again with the test data split. The more advanced way to perform the validation is to split only training and testing, and divide the training split into k folds, where k-1 folds are used for training and the remaining fold is used for validation. The average metric score on all the training-validation fold permutations is used to select the best set of parameters. The two tasks undertaken in this thesis belong to node regression and graph classification. For the node regression, since we want to approximate a real value for nodes of a graph, the normalized root mean squared error is the correct performance metric to use. However, the problem will be transformed into a classification task by defining ranges of values, thus the correct performance metric will be the accuracy or the f1-score averaged over all classes. For the graph classification task, code subroutines classification, there are many classes with imbalance and so a macro average of precision and recall, namely f1-score, is the right performance metric to use.

Evaluation

In this step the best model of each kind is selected. Part of this tasks, when using cross-validation is already done in the modeling part. But it's in this step when results are presented and compared altogether. It is important to stress out that the best model must be chosen by the validation performance metric. The test on the test data split must be used to verify that the model is not over-fitting the validation set, which would indicate that the model will no be able to generalize to unseen data. When the validation performance metric is similar to the test performance metric it indicates that the model should be able to generalize to unseen data.

4 Experiments

The goal of this master's thesis is to apply Graph Neural Network models to different problems to create a novel solution. The idea is to get to know how Graph Neural Networks are used in each situation. Two problems are explored: Girvan-Newman algorithm approximation and compiled code function classification. They correspond to the two main tasks that Graph Neural Network perform with success, semi-supervised learning of nodes on a graph and supervised graph classification.

This section will present the configuration of the experiments performed in the thesis. First, the Girvan-Newman approximation experiment is explained. Then the compiled code function classification experiment is described last. In each case, context, goals and motivation are described first. Then a preliminary test is performed on well-known benchmark datasets to verify the software libraries and setup in use work correctly in the computer without any problem. Then the process of data preparation is explained and finally how the part of training a model is executed is summarized.

4.1 Girvan-Newman algorithm approximation

4.1.1 Experiment overview

Context
The Girvan-Newman algorithm is a clustering based approach to perform community detection. It proposed a divisive algorithm based on edge-betweenness for a graph with undirected and unweighted edges. The algorithm focused on edges that are most "between" the communities and communities are constructed progressively by removing these edges from the original graph. It iteratively isolates groups of nodes of a graph by removing the edges that poses the greater edge betweenness value. The worst-case for time complexity of the edge-betweenness algorithm is $O(m^2n)$ and is $O(n^3)$ for sparse graphs, with m being the number of edges and n the number of vertices.

Goal
The goal of this experiment is to find a novel way to approximate the Girvan-Newman algorithm that is faster than the original.

Motivation
Community detection is a popular task with applications in many areas related to social network analysis. An approximated but faster implementation could impact on tasks that need to detect community on large graphs, like the ones extracted from social networks, citation networks, and the like.

Experiment
To be more precise, the function to approximate will be the computation of the edge betweenness for each edge of the graph in each iteration of the algorithm. Thus, the experiments will focus solely on finding a Graph Neural Network that approximates the computation of the edge betweenness of all edges of a graph with a reasonable performance.

For any node in a graph, node betweenness is defined as the number of shortest paths between pairs of nodes that run through it. The Girvan–Newman algorithm extends this definition to the case of edges, defining the "edge betweenness" of an edge as the number of shortest paths between pairs of nodes that run along it. If there is more than one shortest path between a pair of nodes, each path is assigned equal weight such that the total weight of all of the paths is equal to unity.

There are two main ways to approximate the edge betweenness with Graph Neural Network models.

The first one, using supervised learning for doing edge attribute value regression. A model will be trained to approximate the edge betweenness of all edges of a graph. The final approximated Girvan-Newman algorithm would consist of the original process but replacing the computation of the edge betweenness by the trained model.

The second approach, by using semi-supervised learning, one can compute the edge betweenness of several nodes and then train the model to predict the value for the rest of edges of the graph. In that case, the Girvan-Newman algorithm would, in each iteration, first compute the edge betweenness of some edges in the normal way, then use the model to approximate the edge betweenness of the rest of the edges of the graph. By the nature of the Graph Neural Networks that have attained state-of-the-art performance on semi-supervised learning, the second approach seems the most realistic one.

To give a bit more of context on the second approach to approximate the Girvan-Newman algorithm, the steps would be the following:

- **1. Compute the edge betweenness of some edges of the graph:** One advantage is that we could limit how many edge betweenness's are computed by time. Let's say we allow for 10 seconds edge betweenness computations.
- **2. Approximate the edge betweenness of the rest of edges of the graph:** this steps would use a Graph Neural Network in a semi-supervised and transductive setting. This means that the Graph Neural Network should be trained on the actual graph or similar graphs before using it. It could also have another implication, that the graph neural network training can benefit from having a representative sample of the total population of edge betweennesses of the graph. However, this can only be approximated with some heuristic. For example, one could use for training several edges connected to nodes with a high degree and several edges connected to nodes with a low degree.
- **3. Selection of the most central edge:** this step would gather all edge betweenness values from previous steps and decide which edge to remove from the graph(the one with maximum edge betweenness). The rest of steps of the algorithm would go unchanged.

4.1.2 Dataset

Datasets used

For the Girvan-Newman aproximation experiment, well known datasets available from Pytorch Geometric library are considered and inspected:

- Cora
- Watts-Strogatz random generated graph,
- and Karate-Club.

From all of the graphs in those databases, the ones that show a larger number of nodes and higher edge betweennesses values, with the contraints of the memory limit of the computer used, are selected. The reason is the following, for training a model for predicting the edge betweenness of an edge, some parts of a graph are used as training, validation and others as testing. This implies that a small graph will produce dataset splits that contain few edges and so the model trained on those small subgraphs usually won't be able to learn a good approximation. In the testing and inspection of the graphs, the datasets that contain several small graphs are treated as a big graph with disconnected components. When the dataset contains only one graph, it will usually be big and so the sub graphs splits for training, validation and testing will be enough in size for the model to train properly. The final selection has been to use the CORA dataset.

Data preparation

The first step in data preparation is to compute the actual edge betweenness of each edge of the graph, and save all those in the target (or ground true) vector. To compute the edge betweenness, the Python library NetworkX is used. First, the graph is transformed from the Pytorch Geometric Dataset format to a list of edges on a file on disk. This list of edge is then read by the NetworkX library to transform it into a NetworkX graph object. Then, the edge betweenness function of NetworkX is called. The result is a dictionary (key-value map) with the edge betweenness of each edge.

There are two ways to train a model for the edge betweenness. One way is to do a regression on the value of the edge betweenness, in which case the performance metric would be the normalized root mean squared error. The other way is to group edge betweenness values into groups (or buckets) and then consider each group a class, to be able to train a classification model. In this second way the performance metric will be the accuracy. The classification approach has yielded better results and so is the one selected. For creating the buckets or groups, a series of limit values are chosen, then an histogram of counts for each class is generated. This step is repeated until the histogram shows there is a small class imbalance (all histogram bars are almost same size). The best way to select the bucket limits is to use a function that separates data by percentiles, so that class imbalance disappears. The Python Pandas library function qcut is a good option for this. Finally, the information of the limits of the buckets allows us to transform the vector with true edge betwenness real values into the number of

the bucket they belong to. Thus, this number will be the class or group identifier, that the classifier will learn to predict. For the algorithm to work correctly, the class numbers are assigned from 0 to the number of classes minus 1.

For the setup of the dataset splits, the approach has been to generate small sized training splits, and middle and large sized validation and testing splits. This is to satisfy the semi-supervised learning transductive setting. The total number of edges of the graph is divided in 3 different sizes, after applying some random permutation. The size of the training split is computed as the number of classes time a low number like 20 or 300 depending on the size of the graph. The rest of edges are divided into validation and testing split, with a proportion of 1 to 3 or 1 to 1.

4.1.3 Model training

Since the dataset splits could have a high impact on the performance of the trained model, each training is repeated a number of times, for instance 100 times, and the performance is averaged over all the repetitions. This averaging process would compensate for when a very unbalanced dataset split is generated and a model trained on it.

The type of model used is one that allows us to update the state of the edge attributes in the aggregation and combine phase of the iterative process. PyTorch Geometric python library implements a Message Passing Neural Network [14] that is highly generalizable, called MetaLayer (see [36] and [29]), that allows for updating edge attributes. Based on this architecture, different adjustments on the number of layers, the number of units of layers, and other hyper parameters will be modified during hyper-parameter search.

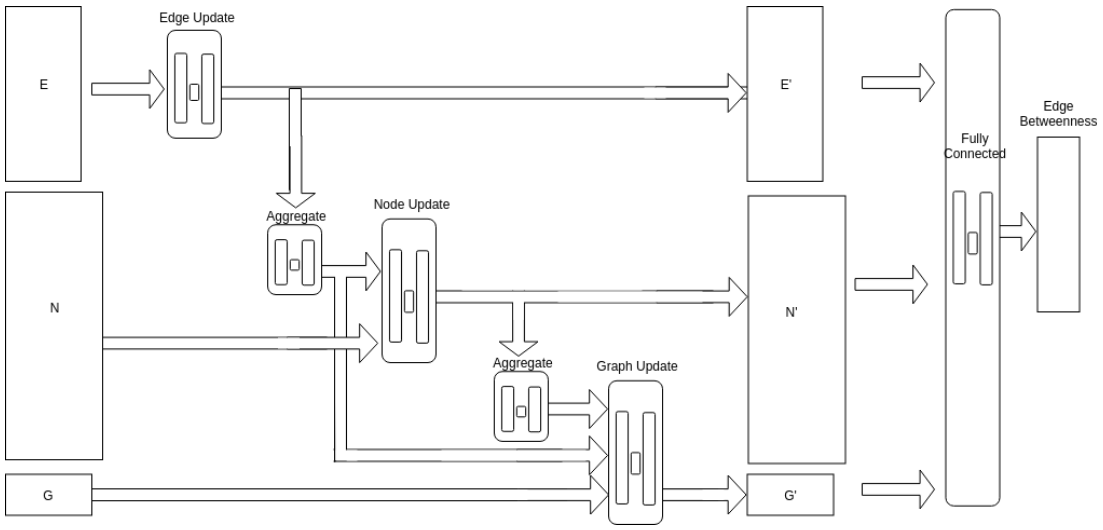

Figure 3: Architecture of the MetaLayer model

The architecture of the MetaLayer model is composed of 4 building blocks. First the node states(also called attributes or features), the edge states and the graph level state are concatenated and fed to the first block: the edge multilayer perceptron(MLP). The edge MLP is a 2 layer neural network with a ReLU in between. The next step is to aggregate the output of the edge MLP per node, concatenate it to the node and graph states and feed this to the node MLP. This second block has 2 MLP networks, one for the aggregation and the other for the combination, each one with 2 layer neural network with a ReLU activation unit in between. Then the edges attributes and node attributes are aggregated globally and the global state of the graph is updated. The output of the MetaLayer is an identical graph with the states (attributes) updated. This new graph stats is fed to 2 fully connected layers that will produce the final edge attributes. The number of units in each layer can be controlled as parameters, which are optimized in an hyper parameter search. Actually this hyper parameter search could be made much more intensive. In this experiment, we don't use the graph level state and graph level state update.

Algorithm 1 Steps of computation in a full GN block.

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots N^e\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right)$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots N^n\}$ **do**
    let $E'_i = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{r_k=i,\ k=1:N^e}$
    $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\rightarrow v}\left(E'_i\right)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right)$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \left\{\mathbf{v}'\right\}_{i=1:N^v}$
  let $E' = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{k=1:N^e}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\rightarrow u}\left(E'\right)$ ▷ 4. Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\rightarrow u}\left(V'\right)$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right)$ ▷ 6. Compute updated global attribute
  return $\left(E', V', \mathbf{u}'\right)$
end function

Figure 4: Architecture of the MetaLayer model (Battaglia et al. 2018 [36])

Evaluation procedure

To evaluate the results, the performance metric used is the accuracy. We could use the F1-score in macro or micro averages, but since the bucketization process is intented to remove class imbalance, the accuracy is still valid.

As a complement to understand how much performance a model attains, the distribution of predicted values versus their true target values is compared in a scatter plot. The cloud of points is plotted in the 2D, to be able to see if it deviates from the 45° line (y=x line). An histogram and a boxplot are also used to inspect the distribution of predicted values versus the target values. All of them are compared with the target values in buckets and also compared with the real values of the edge betweenness

4.2 Compiled code function classification

4.2.1 Experiment overview

Context

There have been some research publications related to the topic of program analysis. As mentioned in the section 2, there have been approaches to classify code snippets [27] and variables [26] and their usage so that a name can be assigned to them.

Goal

Inspired by those examples, the second experiment of this thesis tries to create a predictive model that will classify a snippet of code in assembler language, the programming language that has a direct transcription to machine code in compiled binaries. The experiment will focus on entire functions or subroutines in assembler language. The experiment will not try to replicate any of the conditions of the research that inspired it, since this experiment will be performed on a different programming language than [26] and [27] and a different kind of model than [27].

Motivation

This experiment could build a classification model that is useful for professionals that work on anti-virus companies. In any anti-virus company, there are security engineers that analyze malicious code, also called computer virus or malware. Part of the process of analyzing a malicious code consists in reverse engineering the code contained inside a binary file. Besides fighting against code obfuscation techniques, the reverse engineer is faced with several thousands of assembler subroutines or functions that he needs to inspect to find where the important code functionalities reside. One way to proceed is from the hints of the interaction of the malicious code with its environment. When a malicious code reads files or connects to Internet urls, commonly called artifacts, the reverse engineer can begin to trace back the functions that use those artifacts. In the process of tracing the code related to those artifacts, or in looking for main actions like connection to the Internet, the reverse engineer will write comments, annotate code snippets or rename functions. It is estimated that having an indication of the main functionality that a code snippet performs can help reverse engineers to find more quickly the evidences they need when analyzing malicious code. This indication can be implemented as renaming the functions of a binary. The renaming of a function can be based on the classification that a machine learning model will predict on it.

Experiment

The experiment consists in several complex steps. First, building a dataset that is suitable for training a model on classifying code on a set of predefined classes. Some binaries that contain functions related to the chosen predefined classes are collected from open source software. Next, those binaries and their functions are analyzed by a program that translates machine code to assembler language. This kind of tool is called a disassembler. Then, different representations of those functions in assembler have to be generated: a graph for each function and some features related to the assembler code itself and to the structure of the graph. After that, the collected dataset must be correctly labeled. Since the intent is to train a model in a supervised setting, each function (corresponding to a sample of the dataset) must be labeled with one of the predefined classes. Finally, with the dataset correctly labeled and with all its features in place, the models can be trained to classify assembler code functions. A set of baseline classification models, then some basic NLP models and finally a set of GNN models will be trained.

4.2.2 Dataset

Let's first review the complex data preparation process. There's no open source dataset with the properties needed for the experiment so it will be created from scratch.

Data sources

Since the main audience could be malicious software analysts, one possibility of getting data from would be to collect analyzed malicious code. It turns out that there's not many sources of analyzed malicious code, and also, malicious code is plenty of obfuscation techniques, which would be an entire different experiment to deal with.

Instead, the preferred strategy is to use well-known software that contains the functionality according to the predefined classes. A decision was need to be made on the set of labels that the dataset will contain. A set of main functionalities related to general purpose applications would be:

- network
- cryptography
- disk access
- memory manipulation

According to that list, several open source tools and libraries are collected, in order to have at least several binaries in each of the previous categories. To that end, open source binaries from web servers, email clients, ftp servers and clients, cryptographic libraries, disk utilities and C library are collected. The list of used software is detailed in the Annex A.

To be able to later assign a label to each sample, where each sample is a function from the selected applications, there are a few options available. The first one is to consider all functions of a binary to be labelled with the main purpose of the binary. For example, all functions of a web server would be labelled network. This approach is too coarse grained and therefore may lead to useless classification models. A second approach would consist in obtaining the open source software sources and then compiling them with debugging information. This type of compilation maintains the original function names that developers assigned to functions. Examples of function names are *asn1_print_fsname* and *bio_sock_error*(see Annex A).

Data transformation

For generating the dataset, a plugin in Python programming language has been developed to be used inside a disassembler program, namely the free version of IDA (IDA Freeware version), a popular professional disassembler program but in its freeware version. As introduced in the context of this experiment, a disassembler is a program that reads compiled binaries and converts machine instructions to assembler instructions, and its main purpose is to reverse engineer software for debugging purposes.

This plugin visits all the functions detected in a binary and extracts the information of instructions, operands and cross-references to generate a graph of the relationship between them. Every instruction, operand, memory address or constant is a node of the graph. Every relationship between them is a vertex of that graph. There's worth mentioning that instructions are executed one after another and so usually an instruction will be linked to its previous and next instruction in the execution flow. However, sometimes loops, conditional branches and function calls modify this sequential execution flow by adding extra edges.

The graph is saved in a format suitable for the Python library NetworkX to read it and import it. The format consists of 2 text files, one with the list of nodes and their attributes, the other with the list of edges and their attributes. For more details, see Annex B.

Supervised labels generation

After obtaining the assembler code of each functions and its original name, some simple rules were created to derive a label from the function name in a programatic way. Basically the appearance of keywords would drive this process. Additionally, human supervision has been applied to improve the labeling assignments.

Four different versions of the dataset have been generated:

- v0: each binary and all its functions have the same label. This is a coarse grained and unrealistic approach. It is suspected that the models would learn spurious relations with this dataset.
- v1: 10 different labels have been chosen, and function labels has been assigned by keyword appearance in the debugging information (original function name).
- v2: a list of topic and tasks has been compiled and combined to generate 120 different labels. Keyword appearance rules and human supervision have been used to assign labels.
- v3: a reduced list of topic and tasks has been compiled and combined to generate 24 different labels. Keyword appearance rules and human supervision have been used to assign labels.

The idea was to find the correct granularity of the dataset. Trained models tend to perform better in the smaller sets of labels (v0 and v1), but there is the risk that the model does not capture any of the information of interest. More detail can be found in Annex C.

Feature engineering

Besides the graph representation of each function of a binary file, the attributes that will be used by baseline models are also extracted or generated. There are topological features from the graphs and features extracted directly from the code itself, like the Bag of Words embedding (fined-tuned by TF-IDF). The topological features include the most common statistics on graph structure like diameter, average degree, average clustering, average shortest path length and assortativity coefficient. The code features, besides the Bag of words embedding from the words on the code itself, include number of registers, number of instructions, number of functions called, etc. The complete list can be found in the Annex D.

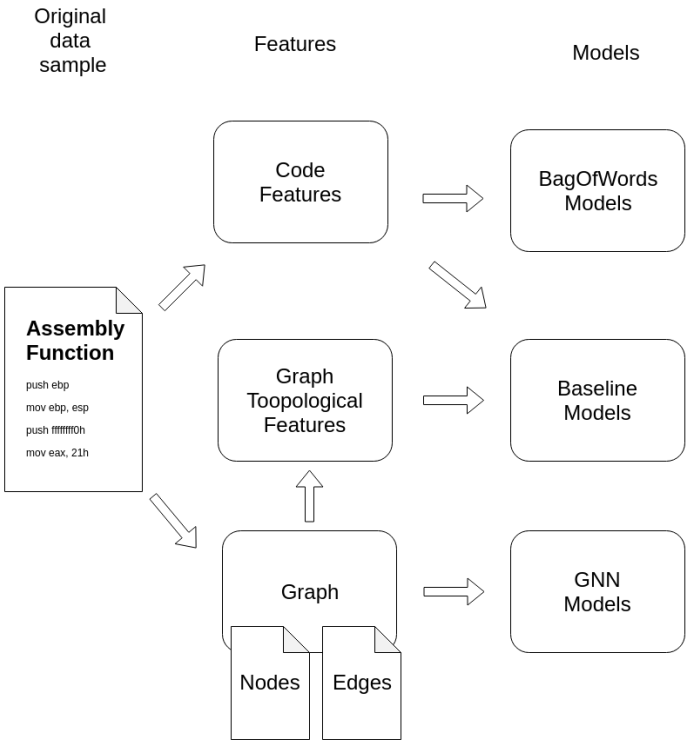

Figure 5: Features exracted from the dataset samples and their relationship with the models trained

All of these features and graph representation will be transformed into a Dataset module from Pytorch Geometric library, [29]. This will allow the training of Graph Neural Network models to use the complete ecosystem of PyTorch Geometric for loading graphs, training models and test their performance. For the baseline models, the topological features and code features will be transformed into tensors of floats. And for the models that use bag of words embedding, the bag of words is not exactly generated in this stage, but it is rather fed by the string that contains all the code listing, during training time, to adjust the words to be taken into account by testing different maximum and minimum frequencies. Once the model is trained, it has also decided the best version of the bag of words embedding. In any case, the output of the bag of words is an embedding that is fed to the downstream machine learning models.

An initial distribution analysis is performed on the different code and topological features to see if the features carry any discriminative power within its signal. The analysis confirms there is a significant amount of signal in the data (D).

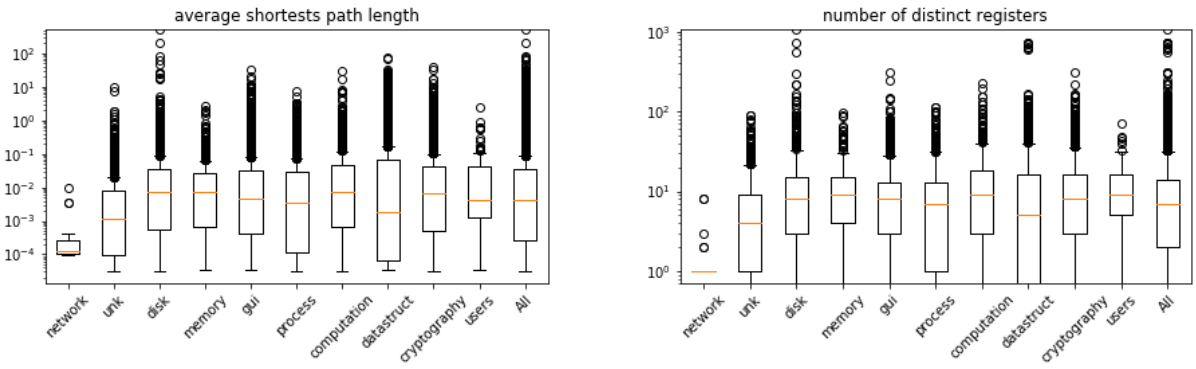

Figure 6: Boxplots of the distribution of the data for several features

4.2.3 Model training and evaluation

Compiled code classification models

A set of trainings will be performed on baseline models, Bag-of-word models and GNNs.

The set of baseline models used are:

- Logistic Regression
- Decision trees
- Random Forest
- XGBoost
- MLP (Multi-layer perceptron)

The set of Bag-of-words models consists of the baseline models of the previous list, but using the bag-of-word tfidf embedding as an input. There will also be mixed features from topological features or code features, concatenated with the the Bag of words tfidf embedding.

The set of Graph Neural Networks used consists of combinations of:

- pooling layers to reduce dimensionality
- GCN, GraphSage, Gated Graph Neural Networks, MetaLayer(Message Passing Neural Network) and Graph Isomorphism Network (GIN)
- global pooling to extract a graph level set of attributes
- some fully connected layers at the end

Based on this architecture, different adjustments on the number of layers, the number of units of layers, and other hyper parameters will be modified during hyper-parameter search.

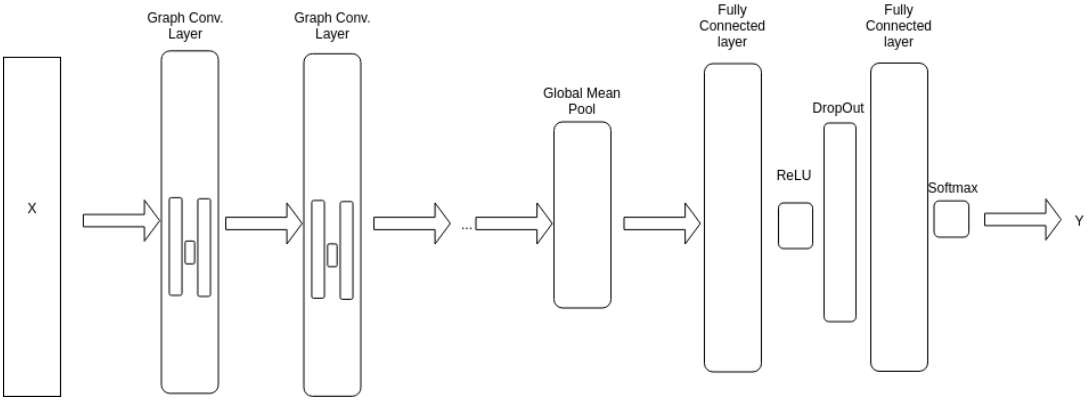

Figure 7: Architecture of the Graph Isomorphism Network (GIN) model

Evaluation

To evaluate model performance, the f1-score with a macro average will be used. F1-score allows to measure precision and recall at the same time (it's the harmonic mean). To tackle the class imbalance present in the data, an undersample strategy on the training set has been applied to re-balance the classes, and then the Macro averages are used as the performance measure. Restoring class balance protects the training of the model from the possibility that it learns to classify only the majority class.

5 Results

This section presents the results and their respective conclusion on all the experiments and their preliminary tests.

5.1 Preliminary tests

As a way of verifying that available code from published research works correctly on the computer on which the experiments will be performed, a published experiment is reproduced. A task of semi-supervised classification is reproduced locally, training a set of GCN, ChebNet, GAT , SG and APPNP models. The Cora dataset is used for this task. The Cora dataset([37] consists of 2708 scientific publications classified into one of seven classes. The goal is to verify that the computer that will support the experiments can train those models without memory problems or other types of problems. Also it is a way to check that some configurations of hyper-parameters yield good results for later comparison with the real experiments. We observe that test accuracy is correct in all the models trained, successfully replicating the experiments. Setup of the environment is also correctly verified. The results of the semi-supervised classification task on the CORA dataset are show on the following table:

Model	Dataset	Tested Accuracy	Reported Accuracy
SGConv	CORA	79.0 ± 0.018	81.7 ± 0.1
ChebConv	CORA	76.9 ± 0.027	81.4 ± 0.7
APPNP	CORA	80.8 ± 0.017	83.3 ± 0.6
GATConv	CORA	80.3 ± 0.016	83.1 ± 0.4
GCNConv	CORA	78.9 ± 0.018	81.5 ± 0.6

Table 1: Reproduced benchmark semi-supervised classification experiments on CORA dataset with originally reported accuracy [29]

A reproduction of some of the results in graph classification tasks by Graph Neural Networks are replicated in this preliminary tests section to make sure that we are able to attain a similar performance with the tools at hand (Pytorch Geometric implementation of Graph Neural Network components). The graph classification tasks on the datasets Proteins and REDDIT binary are reproduced with different model architectures than the ones on the benchmarks of Pytorch Geometric library. We observe that test accuracy is correct in all the models trained, successfully obtaining accuracies close to the ones in the experiments ([29], [38]). Setup of the environment is also correctly verified. The results of the graph classification task on the different datasets are shown on the following table:

Model	Dataset	Test Accuracy	Reported Accuracy
GGNN1	REDDIT-BINARY	71.30%	74.9% to 92.1%
GGNN1	PROTEINS	76.23%	72.33% to 84.08%

Table 2: Reproduced benchmark supervised graph classification experiments

5.2 Girvan-Newman approximation

Main experiment

As detailed in the experiments section, the main experiment is to train a model to approximate the edge betweenness of part of the edges of a graph, in a semi-supervised manner. The performance is measured with the accuracy and F1-score macro averaged after transforming the edge betweenness values into a finite number of ranges (bucketization or discretization). Different number of discrete ranges have been tested and using 28 ranges is the most optimal. For generating the buckets(discrete ranges) the function qcut from Pandas Python library has been used to split the data by the desired number of percentiles. The results are shown in the following table:

Model	Paramteres	RunsEpochs	Splits	Loss	Accuracy	F1-macro
META1	d19d16h10e19n16n15	100-20	100-2000-10556-	2.3561	0.280 ± 0.061	0.251 ± 0.060
META1	d19d16h10e19n16n15	100-250	100-2000-10556-	3.8995	0.139 ± 0.018	0.128 ± 0.020
META1	d19d16h10e19n16n15	100-20	20-500-1500-	3.0269	0.142 ± 0.027	0.127 ± 0.029
META1	d19d16h10e19n16n15	100-250	20-500-1500-	4.9482	0.093 ± 0.013	0.087 ± 0.017
META1	d19d16h10e19n16n15	100-20	200-3000-10556-	2.0677	**0.342 ± 0.078**	**0.312 ± 0.076**
META1	d19d16h10e19n16n15	100-250	200-3000-10556-	3.3414	0.219 ± 0.027	0.208 ± 0.028
META1	d19d16h10e19n16n15	100-600	200-3000-10556-	3.7691	0.142 ± 0.037	0.124 ± 0.039
META1	d19d16h10e19n16n15	100-2	20-500-10556-	3.0165	0.112 ± 0.047	0.088 ± 0.052

Table 3: Edge betweenness approximation with a graph neural network experiment using 28 discrete ranges

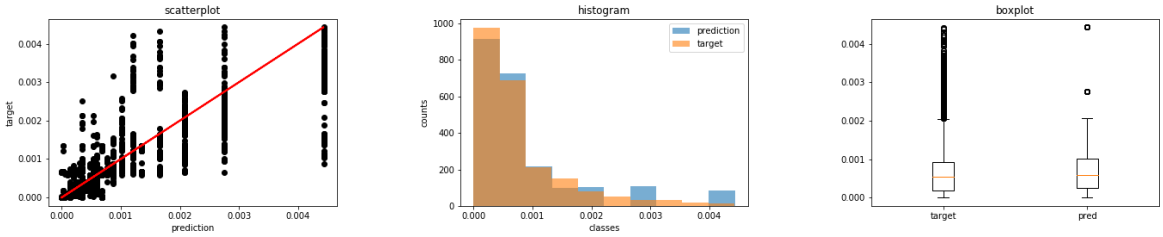

Figure 8: Predictions compared to target values in an approximation of the edge betweenness using 28 discrete ranges and 20 epochs of training (best accuracy model)

The best model is the one trained for 20 epochs, with a dataset split of 200, 3000 and 10556 train, validation and testing number of samples respectively. It attains a 34.3% of accuracy as well as a 31.2% f1-score macro averaged. However, when looking into the scatter plots of predicted values versus target values, a trained model with 250 or 600 epochs seems to be better fitting the real values on some of the runs. For each model we want to train, given that the dataset split is critical to the model performance, we must run it for several times (100 in this case) and average the performance metrics to get a more realistic model performance. In the end, even if for some dataset splits the models fitted have a better performance, it is the average performance over all the splits that counts. We want the model to be able to generalize.

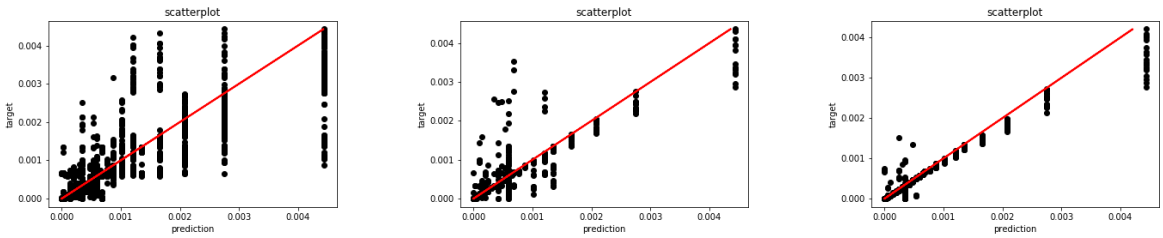

Figure 9: Predictions compared to target values in an approximation of the edge betweenness using 28 ranges with 20, 250 and 650 epochs of training

Conclusion

The best trained model for predicting the edge betweenness obtains a 34.3% of accuracy as well as a 31.2% f1-score macro averaged in the 28 discrete ranges setup.

One important negative aspect of the models trained so far in this experiment, is that the predictive model will not predict which edge has maximum value of edge betweenness but a whole group or bucket. The final goal of the experiment is to use the model for the Girvan-Newman algorithm which needs to find the edge with highest edge betweenness. Since the model uses buckets, all edge betweennesses predicted to belong to a bucket will have all the same value.

For the task at hand, the most important predictions are those of the edges with high edge betweenness. In that aspect, the more granular model (the one with 28 discrete ranges) is more precise and the high value predictions are less noisy than other more coarse grained models tested. A more granular model, would return a smaller set of edges that are predicted to have the highest edge betweenness, and so the subsequent tasks to refine this prediction would be faster.

Improvements

The first basic improvement that could be looked into is to perform a longer hyper parameter (and data partitioning configuration) search, to see if this kind of model can obtain a better accuracy.

Also, it would be important to test its performance on several heterogeneous graphs, to get an idea of how stable this model training is.

To improve the usability for the Girvan-Newman approximation, the precision in high values should be increased. One naive solution would be to give the samples in the bucket their maximum value, because the problem is to find the real maximum and not to increase the overall values of a bucket. In one hand, to have the model to predict higher values, a smaller range with the most higher values of the training set could be created. But, since this would introduce a great deal of class imbalance, the performance of the model would decrease. This trade-off should be tested thoroughly. A good approach to solve the class imbalance would be to use a weighted loss function during the training of the Graph Neural Network. This loss would have a weight associated with the inverse of the number of samples a bucket contains. Another thing to keep in mind, is that the loss function and the error function(metric score) should be changed to something more beneficial for finding the maximal values, for example to use the f1-score of the class with highest edge betweenness value.

One different approach to this problem would be to recompute the real edge betweenness of edges predicted to belong to the highest bucket. Depending on the size of the higher bucket, this process could harm the speed of the approximation of the Girvan-Newman algorithm.

Finally, for the Girvan-Newman algorithm, an inductive setup of the graph neural network algorithm should be tested. The current transductive setup (semi-supervised learning where a few edge betweennesses will be actually computed to quickly train the model and the rest will be predicted) might impact the speed of execution. Since in each iteration some edge betweenness need to be computed, then the model needs to be trained (or fine-tuned) and then this same model is used to predict the rest of edge betweenness's, the process could be slower than the original algorithm. If a model is trained once and can directly be used in each iteration, the overall process should be faster. There is a middle ground solution, where the graph neural network model is only trained in the very first iteration of the Girvan-Newman algorithm and then in the subsequent steps it is just used for prediction.

5.3 Compiled code function classification

Main experiment

The next results tables show the performance of baseline and Graph Neural Network models on the code function classification dataset in versions v1 and v3. The results are sorted by the obtained cross-validation score, in order to choose the best model without overfitting on the test set. The test set scores are just shown to verify that the validation scores are consistent and not overfit on the validation dataset split.

Model	Features	CV F1-macro	Precision macro average	Recall macro average	F1-macro
MLP2	Code Bow and topological	0.572	0.350	0.291	**0.299**
MLP4	Code Bow and topological	0.528	0.252	0.251	0.237
MLP2	Topological and code attr.	0.420	0.232	0.203	0.186
XGBoost	Code Bow and topological	0.387	0.470	0.374	**0.401**
Random Forest	Code Bow and topological	0.307	0.297	0.293	**0.290**
Random Forest	Topological and code attr.	0.186	0.246	0.198	0.186
XGBoost	Topological	0.171	0.203	0.190	0.190
Random Forest	Topological	0.164	0.199	0.187	0.186
GIN	Graph	0.147	0.122	0.180	0.145
GGNN1	Graph	0.134	0.052	0.098	0.065

Table 4: Function classification experiment results with dataset v1

The models with best performance are

- an XGBoost with a gbtree booster and a maximum depth of 150,
- a Random forest with 4 estimators and a maximum depth of 150,
- and a Multi-layer perceptron with 3 fully connected layers with 94, 20 and 15 hidden units.

Those 3 best models are trained using a version of the dataset with the Bag-of-words embedding and the topological features. The best graph neural network model is a Graph Isomorphism Network with 4 layers and 128 hidden units plus a fully connected layer, trained only with the graph of each sample as the dataset, and its performance is the worst in that case. See the annex E for the exact parameters of the models and their training steps.

Model	Features	CV F1-macro	Precision macro average	Recall macro average	F1-macro
MLP2	Code Bow and topological	0.395	0.224	0.222	**0.213**
MLP4	Code Bow and topological	0.326	0.141	0.187	0.156
MLP3	Code Bow and topological	0.313	0.184	0.176	0.151
RandomForest	Code Bow and topological	0.208	0.288	0.219	**0.209**
XGBoost	Topological and code attr.	0.205	0.236	0.218	**0.221**
GIN	Graph	0.199	0.192	0.205	0.198
Random Forest	Topological and code attr.	0.179	0.196	0.181	0.178
Logistic Regression	Code Bow and topological	0.173	0.223	0.202	0.188
XGBoost	Topological	0.108	0.112	0.116	0.112
RandomForest	Topological	0.102	0.097	0.104	0.098
GGNN1	Graph	0.064	0.106	0.103	0.080

Table 5: Function classification experiment results with dataset v3

The models with best performance are:

- an XGBoost with a gbtree booster and a maximum depth of 150,
- a Random forest with 16 estimators and a maximum depth of 16,
- and a Multi-layer perceptron with 3 fully connected layers with 96, 20 and 15 hidden units.

One remarkable difference in this results table, is that the best performing model on the test set does not use the Bag-of-words embedding. The other 2 best models are trained using a version of the dataset with the Bag-of-words embedding and the topological features. The best graph neural network model is a Graph Isomorphism

Network with 2 layers and 64 hidden units plus a fully connected layer, trained for 500 epochs only with the graph of each sample as the dataset, and its performance is close to the top 3 models. See the annex E for the exact parameters of the models and their training steps.

Results with the v2 dataset are not shown as the models where not performing better than random and their training took much more time than with v1 and v3 dataset versions.

Conclusion

As a conclusion, it has been proven that Graph Neural Network models can be used to classify snipets of assembler code by their corresponding graph of instructions, with a similar performance as the most common machine learning classification models. However, their performance is not the best one, it seems that the Graph Neural Network models trained in this experiment lack some capacity when compared to the models with better performance. But to be completely fair, the Graph Neural Network models have attained similar performance as the baseline models that only use information form the graph. The baseline models that have a higher performance are using the semantic information derived from the code.

Thus, an important result is that the amount of information with discriminative power that the dataset contains is the most influential aspect for obtaining a good classification model. The signal carried by the different features of the dataset is not the same. The semantic similarity information contained in the Bag-of-words embedding is the most discriminative feature in this experiment. It is more powerful than the isomorphism detection that is performed by Graph Neural Networks on the graph, or the topological features extracted from it. The different transformations of the original dataset, version v1 with 10 classes, version v2 with 120 classes and v3 with 23 classes also influence the final performance, the way the models are trained and the amount of resources they require. The version v1 and v3 turned out to be the most suitable versions for this tasks and the resources at hand.

About the Graph Neural Networks, one key take aways is that their training is complex and delicate. Not all hyper-parameter configuration yields and acceptable result, and furthermore, the partition of the dataset plays a big role in the final performance of the model trained. It is advisable to use a higher number of folds in the cross-validation process to get a better idea of the generalization power of the model that is going to be selected. It is not easy to find the best configurations across the different dataset splits, because the search space is huge and the time it takes for training those models is high. They could be compared to training SVMs, big CNNs and LSTMs. Nonetheless, it seems that well trained Graph Neural Networks can attain similar performance than the best baseline models when using only information derived from the graph or the graph itself.

Improvements

As stated in the conclusion, the discriminative power of the semantic similarity that can be computed from Bag-of-word embeddings seems to be higher than the one derived from the graph itself. So the obvious next step is to include the semantic information into the graph or to combine it with the Graph Neural Network output embedding into the final fully connected layers to build a classifier that outperforms the baselines models shown so far. The former approach could be the most powerful and the latter seems to be the easiest one to implement.

A key aspect to improve in this second task is the dataset labeling. The labels of the data set had to be created from scratch, and this procedure might have room for improvement. Also the dataset is very noisy and has a lot of class imbalance. Improving the dataset class distribution, hopefully with more samples could also be studied. One take away of this second experiment is that this task was too ambitious, and probably a more focused approach could be better. For example the model could be trained for classifying snippets of code with a certain maximum length in order to detect loops, branches and function calls as a building block for later detecting bigger compounded code structures.

6 Conclusion

The goal of this master's thesis was to explore some of the most common Graph Neural Network models and their applications. It has been a real pleasure to learn about this novel techniques and their internals. The applications of those models are wide and exciting, even though it must be stressed that they are not general purpose machine learning algorithms. But in the end, a lot of data can be expressed in a graph structure. The feeling is that these types of models will continue to be explored and become very successful in some areas.

As part of the research of this master's thesis, two different tasks have been tackled with Graph Neural Networks, approximating the Girvan-Newman clustering algorithm and building a classifier for compiled code functions renaming.

The first experiment aimed at an exciting modification of a powerful clustering algorithm, the Girvan-Newman algorithm. The task was to train a Graph Neural Network for edge attribute classification. Many interesting approaches have been applied to make the model obtain a decent performance. The model does approximate the edge betweenness in a transductive setting. This means that in each iteration of the Girvan-Newman algorithm it could be used in combination to computing the real edge betweenness of some of the edges of the graph. The goal was to discover or prove that this approximation was possible, by showing the performance of the model on computing the edge betweenness.

The next steps that could be implemented are to perform more experiments to obtain a more precise prediction for the edges with highest edge betweenness in order to make the model more suitable for finding the edge with highest centrality. It also would be interesting to train this model in an inductive setting, where it could directly be used in all the iterations of the Girvan-Newman algorithm without retraining and for all edges. Finally, testing the different ways to combine this model that approximates the value of the edge betweenness and the Girvan-Newman model is a very interesting task, where the main point would be to find a good balance between overall time of the algorithm and precision of the approximation.

The second experiment aimed at building a model that could be applied code analysts and reverse engineers work. It consists of a classification model for code fragments(subroutines or functions) from a compiled code binary. The class prediction could then be used to rename the functions and speed up the process of debugging or analyzing malicious unknown code. The task has been approached with basic machine learning models as well as Graph Neural Network models. Part of the excitement of this tasks was to represent a list of instructions in assembler language in a graph structure and also extract features related to it. Different types of machine learning models have been trained. The Graph Neural Network models were able to attain same performance as baseline models using only graph related information. They have not been able to surpass the performance of the baseline models that used the bag-of-words embeddings derived from code itself. We conclude that the bag-of-words embedding carries more discriminative power than the graph derived from code or their topological features. Another take away is that the training of Graph Neural Networks is more costly and more complex than for other basic machine learning models. The highest impact on the performance of the trained model is caused by the way the split of the training, validation and testing dataset is chosen. Also the hyper-parameters have to be chosen carefully to avoid the performance to drop significantly.

Improvements have been proposed to this task, consisting of combining the semantic similarity information carried in the bag-of-words embeddings with the graph information to train a more powerful Graph Neural Network. The combination could be done at the node or at the graph level. Another improvement proposed is to review the dataset generation and labeling in order to generate a less noisy dataset, probably changing the focus to a smaller set of classes but that could yield a model with better performance.

References

[1] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[2] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *Proceedings of the International Conference on Learning Representations*, 2017.

[3] W. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[4] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[5] W. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *Advances in Neural Information Processing Systems*, 2017.

[6] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv:1810.00826*, 2019.

[7] Zonghan Wu, Shirui Pan Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *arXiv:1901.00596*, 2019.

[8] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *International Conference on Learning Representations (ICLR)*, 2016.

[9] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. *In Proceedings. 2005 IEEE International Joint Conference on Neural Networks*, 2:729–734.

[10] L. B. Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. *Proceedings of the International Conference on Neural Networks*, 2:609–618, 1987.

[11] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral net- works and locally connected networks on graphs. *Proceedings of International Conference on Learning Representations*, 2014.

[12] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[13] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1(2):3, 2017.

[14] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *Proceedings of the International Conference on Machine Learning*, pages 1263–1272, 2017.

[15] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734, 2014.

[16] X. Shen, S. Pan, W. Liu, Y.-S. Ong, and Q.-S. Sun. Discrete network embedding. *Proceedings of the International Joint Conference on Artificial Intelligence*, 7:3549–3555, 2018.

[17] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations,. *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining.*, pages 701–710, 2014.

[18] Danfei Xu, Yuke Zhu, Christopher B Choy, and Li Fei-Fei. Scene graph generation by iterative message passing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5410–5419, 2017.

[19] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *arXiv preprint arXiv:1801.07829*, 2018.

[20] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

[21] Federico Monti, Michael Bronstein, and Xavier Bresson. Geometric matrix completion with recurrent multi-graph neural networks. In *Advances in Neural Information Processing Systems*, pages 3697–3707, 2017.

[22] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983. ACM, 2018.

[23] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702, 2017.

[24] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

[25] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 140–151. ACM, 2019.

[26] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[27] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[28] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[29] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[30] Girvan, Michelle, and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[31] Bedi, Punam, Sharma, and Chhavi. Community detection in social networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 6:n/a–n/a, 02 2016.

[32] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the national academy of sciences*, 101(9):2658–2663, 2004.

[33] Matthew J Rattigan, Marc Maier, and David Jensen. Graph clustering with network structure indices. In *Proceedings of the 24th international conference on Machine learning*, pages 783–790. ACM, 2007.

[34] Jingchun Chen and Bo Yuan. Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

[35] Mark EJ Newman. Analysis of weighted networks. *Physical review E*, 70(5):056131, 2004.

[36] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[37] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[38] Karsten M Borgwardt, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1):i47–i56, 2005.

[39] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems 29*, 2016.

[40] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv:1811.05868*, 2019.

[41] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

[42] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

[43] Peter D Hoff, Adrian E Raftery, and Mark S Handcock. Latent space approaches to social network analysis. *Journal of the american Statistical association*, 97(460):1090–1098, 2002.

[44] José Suárez-Varela, Sergi Carol-Bosch, Krzysztof Rusek, Paul Almasan, Marta Arias, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Challenging the generalization capabilities of graph neural networks for network modeling. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 114–115. ACM, 2019.

[45] Paul D Dobson and Andrew J Doig. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of molecular biology*, 330(4):771–783, 2003.

Appendices

A Compiled code binaries

To perform the experiment on function renaming, the dataset used was a set of well-known open source programs:

- Apache http server
- Filezilla ftp client
- Nginx http server
- OpenSSL library
- GlibC library

They were compiled from source using debugging options that make the final binary include the function names (also called symbols). Basically, each package or software had different options to include in the compilation command, but most of the times it required to include the -d for debugging option followed by some other indications.

Once the code is compiled with debugging information, the different functions or subroutines can be related to a function name that is listed inside the binary. Examples of function names are the following:

- aes_encrypt_cbc, which is highly informative of the purpose of the function, encrypting with AES algorithm using Cipher Block Chaining.
- apr_socket_sendfile, which is again highly informative telling that it sends a file through a socket
- _zn9__gnu_cxx12__pool_allocice10deallocateepcy, others are not so informative or easy to understand, but at least it shows that it is handling memory pool allocations.

This will be the base on which a ruleset will transform each of the function names (a total of more than 30000) into a small set of classes (8 for v1, 170 for v2 and 24 for v3 versions of the dataset)

B Feature extraction details

This annex will present the details of the implementation of the feature extraction from the dataset of assembly function code listings, used in the task of assembly function code classification.

Graph of an assembly function code

In this thesis, the tasks that classifies binary code functions takes advantage of programs that are prepared for reading and parsing binary files into human readable formats, using a programing language called assembler or assembly. However, a binary file is a file that contains bytes organized following a convention, or file format like PE or COFF, completely dependent of the operating system, but not assembler directives. The bytes contained in those file formats represent machine language code. This is not a problem as, per wikipedia definition, an assembly language (or assembler language), often abbreviated asm, is any low-level programming language in which there is a very strong correspondence between the program's statements and the architecture's machine code instructions.

For analysing a binary file, programs that convert the bytes of the binary file, that represent machine code, into assembler instructions are called disassemblers. For this thesis the selected disassembler program is called IDA free, which is a free version of the corresponding professional program IDA pro. This program reads the binary file bytes, according to the corresponding file formats, like PE for Microsoft Windows or COFF2 in Unix operating systems, and is able to interpret the machine code and translate it into assembler code instructions. There is a direct correspondence between machine code and assembler directives.

The selected disassembler program, IDA free, allows for executing scripts written in the programming language python. This scripts are run from inside its user interface to do tasks, like inspection and modification of the database of the binary code loaded.

The python script is ran as a script, also known as "plugin", inside the disassembler program (IDA free). It takes advantage of series of exposed interfaces or methods that read the database that contains the information of the disassembled binary file and its instructions. The procedure followed by the python script is to process all the lines of code of each function contained in the binary file. For each function or subroutine that the disassembler has detected, the python script will write 2 separate text files that contain the list of elements found as nodes and the list of relationships between them as edges, conveniently following a file format that the python package NetworkX can read and import into a NetworkX graph representation. The elements considered are all the types of elements that can appear in assembly code and that the disassembler detects: instructions, registers, memory addresses, immediate values, displacements and called functions. The edges represent relationships between them. For example, the instruction "mov eax, 0x0457AB" is related to the register "eax", the memory address "0x0457AB", as well as the previous and next instruction found in the code listing when following its execution flow. When the instruction is calling another function, for example "call another_function_address", an edge relating this instruction and the function another_function_address must exist.

```
sub_454740      proc near               ; CODE XREF: sub_454770+6C↓p
                                        ; sub_454770+E1↓p
                push    esi
                xor     esi, esi

loc_454743:                             ; CODE XREF: sub_454740+15↓j
                mov     eax, [edx+esi*4]
                test    eax, eax
                jz      short loc_454753
                cmp     eax, ecx
                jz      short loc_45475B
                inc     esi
                cmp     esi, 200h
                jl      short loc_454743

loc_454753:                             ; CODE XREF: sub_454740+8↑j
                xor     eax, eax
                pop     esi
                retn
; ---------------------------------------------------------------------------

loc_45475B:                             ; CODE XREF: sub_454740+C↑j
                mov     eax, 1
                pop     esi
                retn
sub_454740      endp
```

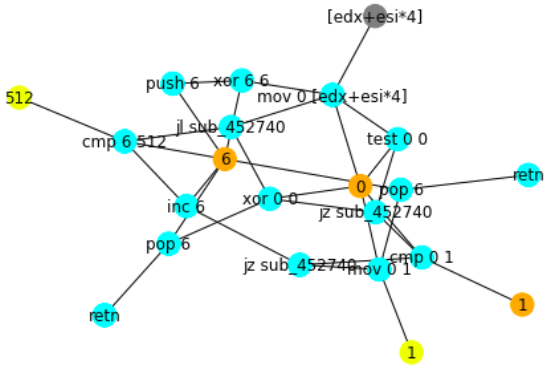

Figure 10: Assembler code transformed into a graph representation

Several assembly code functions are selected for testing the python plugin that generates the graph of the function code. The procedure is simple, the assembly code is printed as a code listing (each instruction is shown in assembly language in a separate line) like the disassembly program shows it inside its user interface. Then the contents of the file containing the description of the nodes, usually called function_name_nodes.txt where the function_name part is replaced by the actual function name being analysed, and the file containing the set of edges joining nodes, usually called function_name_edges.txt where the function_name part is replaced by the actual function name being analysed, are compared to the code listing to verify that every line of code, every register, memory address and the like is present in the nodes file, and every relationship between instructions, registers, memory addresses, called functions and the like is also present in the edges list file.

```
4532064 r6 {'type': 'register'}
4532064 4532065 {'type': 'instr'}
4532064 4532059 {'type': 'instr'}
r6 4532033 {'type': 'register'}
r6 4532046 {'type': 'register'}
r6 4532032 {'type': 'register'}
r6 4532057 {'type': 'register'}
r6 4532047 {'type': 'register'}
4532053 4532035 {'type': 'unkown'}
4532053 4532047 {'type': 'instr'}
4532053 4532055 {'type': 'instr'}
4532035 4532033 {'type': 'instr'}
4532035 4532038 {'type': 'instr'}
4532035 r0 {'type': 'register'}
4532035 p4 {'type': 'phrase'}
4532033 4532032 {'type': 'instr'}
4532057 4532058 {'type': 'instr'}
4532057 4532055 {'type': 'instr'}
4532042 r1 {'type': 'register'}
4532042 r0 {'type': 'register'}
4532042 4532040 {'type': 'instr'}
4532042 4532044 {'type': 'instr'}
r0 4532055 {'type': 'register'}
r0 4532059 {'type': 'register'}
r0 4532038 {'type': 'register'}
4532040 4532038 {'type': 'instr'}
4532040 4532055 {'type': 'unkown'}
4532044 4532059 {'type': 'unkown'}
4532044 4532046 {'type': 'instr'}
4532059 im1 {'type': 'immediate'}
4532046 4532047 {'type': 'instr'}
4532047 im512 {'type': 'immediate'}
('4532064', {'type': 'instr', 'content': 'pop 6'})
('r6', {'type': 'register', 'content': '6'})
```

```
('r6', {'type': 'register', 'content': '6'})
('r1', {'type': 'register', 'content': '1'})
('r0', {'type': 'register', 'content': '0'})
('im1', {'type': 'immediate', 'content': '1'})
('p[edx+esi*4]', {'type': 'phrase', 'content': '[edx+esi*4]'})
('im512', {'type': 'immediate', 'content': '512'})
('4532032', {'type': 'instr', 'content': 'push 6'})
('4532033', {'type': 'instr', 'content': 'xor 6 6'})
('4532035', {'type': 'instr', 'content': 'mov 0 [edx+esi*4]'})
('4532038', {'type': 'instr', 'content': 'test 0 0'})
('4532040', {'type': 'instr', 'content': 'jz sub_452740'})
('4532042', {'type': 'instr', 'content': 'cmp 0 1'})
('4532044', {'type': 'instr', 'content': 'jz sub_452740'})
('4532046', {'type': 'instr', 'content': 'inc 6'})
('4532047', {'type': 'instr', 'content': 'cmp 6 512'})
('4532053', {'type': 'instr', 'content': 'jl sub_452740'})
('4532055', {'type': 'instr', 'content': 'xor 0 0'})
('4532057', {'type': 'instr', 'content': 'pop 6'})
('4532058', {'type': 'instr', 'content': 'retn'})
('4532059', {'type': 'instr', 'content': 'mov 0 1'})
('4532064', {'type': 'instr', 'content': 'pop 6'})
('4532065', {'type': 'instr', 'content': 'retn'})
```

Figure 11: Edges and nodes list text files used to convert assembler code listings into graphs

C Labeling of the dataset

In this annex, a description of the process of labeling of the dataset is given.

The task of renaming functions in assembler code will be performed on a dataset of binaries that have been compiled with debugging information, that is with the so called symbols, which are the names of the functions or subroutines contained in the program. So, in the dataset every function keeps its original name.

The goal of the task is to predict a function name, in a more general way, by giving a name that describes or is a synonym of that main task performed by the function being renamed. The current labels (the original names of the functions) are too specific. Thus, we need to assign as a label a name that is much more generalistic, in the sense that it indicates what is the main purpose of the function. The following steps have been followed.

First, a series of topics have been chosen:

- cryptographic
- networking
- graphical user interface
- disk
- data structures
- memory
- users
- computation

Given the type of program binaries used as a dataset (network servers, clients and cryptographic applications), those topics cover the main functionalities of them.

Second, to be able to give a richer sense to the function names, a series of types of actions have been chosen:

- configure
- verify
- save
- delete
- update
- create
- get
- answer
- read
- write
- send
- receive
- show
- hide
- start
- stop
- sync
- parse
- match
- encrypt
- data
- file
- compute
- work

They have been chose based on the visual observation the real names in the dataset and by common sense also.

Third, 2 labeling strategies are created, one that consists of applying only the topic and a more comprehensive that assigns a topic and a task to a function. For example, the first labeling strategy may rename the function aes_encrypt as cryptography, while the second strategy may rename it as cryptography_encrypt. During the training of the models, those 2 labeling strategies will be compared. It is desirable that the second strategy is implemented over the first one, but the chance of success in this second strategy is lower as there are much more classes than in the second one.

Finally, the labels have been assigned following very simple rules based on the real name of each function in the dataset and the number of appearances of each topic and task within it. The script uses the topics and tasks and their synonyms. It counts how many times each topic and each task appears in the real name of the function. The topic and the task that have the maximum number of appearances are assigned to the function. Two dataset are build, one with the first labelling strategy (assigning only the topic) and one with the second strategy (topic and task). The first dataset has 10 classes (9 + unknown). The second dataset has 170 classes.

The final set of labels for each dataset version is shown below:

For the dataset v1:

- gui
- network
- disk
- cryptography
- datastruct
- memory
- process
- users
- computation

For the dataset v2:

- gui-hide
- gui-show
- gui-config
- gui-work
- gui-update
- gui
- network-send
- network-receive
- network-config
- network-save
- network-data
- network-file
- network-verify
- network-answer
- network-get
- network
- disk-work
- disk-read
- disk-write
- disk-delete
- disk-verify
- disk-parse
- disk-match
- disk
- cryptography-work
- cryptography-compute
- cryptography-send
- cryptography-encrypt
- cryptography-verify
- cryptography-config
- cryptography
- ...

For the dataset v3:

- gui-config
- gui
- network-send
- network-parse
- network-config
- network
- disk-file
- disk-read
- disk-write
- disk
- cryptography-encrypt
- cryptography-config
- cryptography
- datastruct
- memory-config
- memory-read
- memory-write
- memory
- process-update
- process-sync
- process-config
- process
- users
- computation

Figure 12: Dataset labelling versions

D Feature engineering in assembly code classification

Some of the machine learning models for classifying code functions in assembly (compiled binaries), make use of features that have been designed outside of the algorithm. It's the case of the code features, those related to aspects of an assembly function code like number of instructions or number of called functions, and the topological features of the graph of a function, those derived from the function's graph based on the relations between instructions, registers, memory addresses and other instructions. For the rest of the models of this task, the ones based on graph neural networks, there is no feature engineering, as they use the graph as an input to learning and prediction.

The code features are aimed at summarizing or extracting relevant aspects of the underlying code. To this goal, counts of number of instructions, registers, memory addresses, function calls and the like are extracted from each assembly function code listing. The full assembly code listing is also extracted as a feature for models that use Bag of Words strategies. These features are simply obtained by reading the text files that list the nodes and the edges of the graph of the function, as the allow for recovering the basic elements on which the graph is constructed, the nodes, which are representations of instructions, registers, memory addresses and the like. The list of features from code is the following:

- number of registers
- number of distinct registers
- number of instructions
- number of displacements
- number of immediates
- number of memory addresses
- number of functions called
- the list of words used in the code listing
- the list of words but replacing any register by the word register (and the same for memory addresses)
- a list of registers used
- a list of functions used

The code features extraction implementation is tested on several assembly function code listings, by comparing the counts of instructions, registers, memory addresses and the like versus those values extracted by the code that reads the text files that list nodes and edges of the graph of the assembly function.

The features that are derived from the topological aspects of graphs, like assortativity and clustering coefficient, are obtained with the implementation included inside the NetworkX Python library.

No testing of these topological features is performed, as the Python library NetworkX is used by a large community and has a great reputation. The list of the topological features extracted is the following:

- number of nodes
- diameter of the graph
- radius
- average degree
- density
- node connectivity
- average clustering
- average shortest path length
- degree assortativity coefficient
- degree perason correlation coefficient

A distribution analysis is performed on all the features to see if they carry a considerable amount of discriminative power towards the different classes of the dataset. The analysis for dataset v1 is the following:

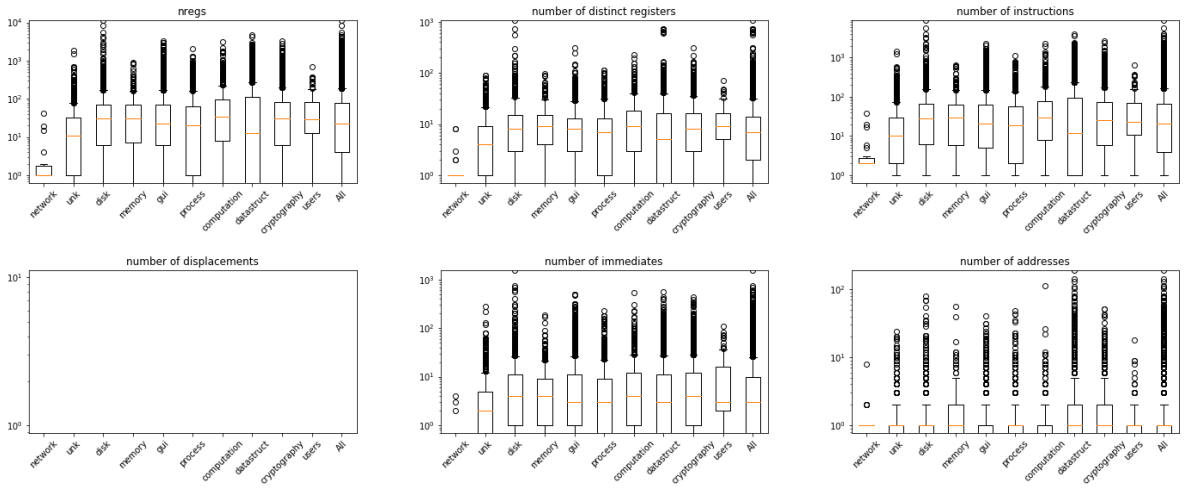

Figure 13: Boxplots of the distribution of the data for several features on v1 dataset

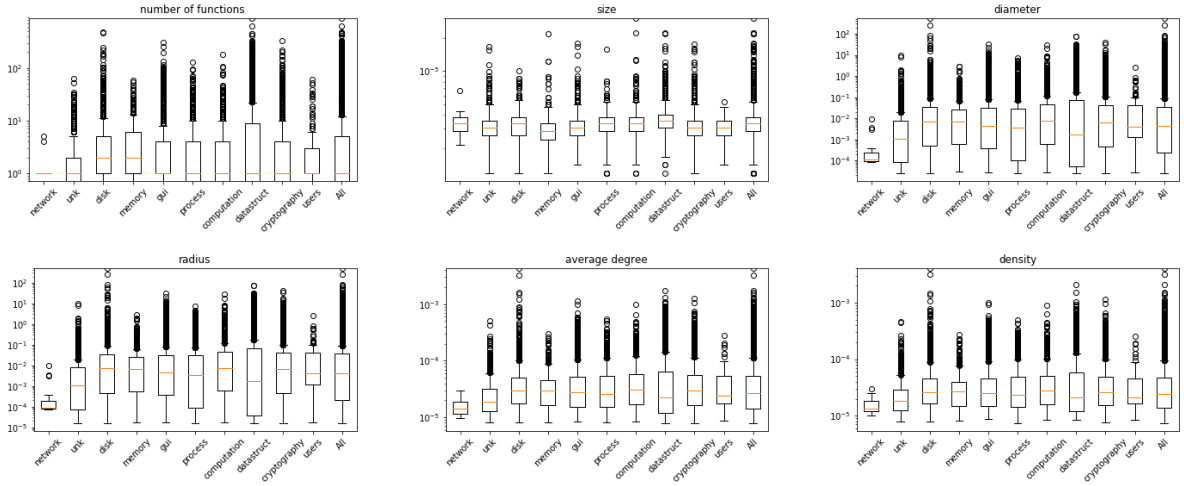

Figure 14: Boxplots of the distribution of the data for several features on v1 dataset

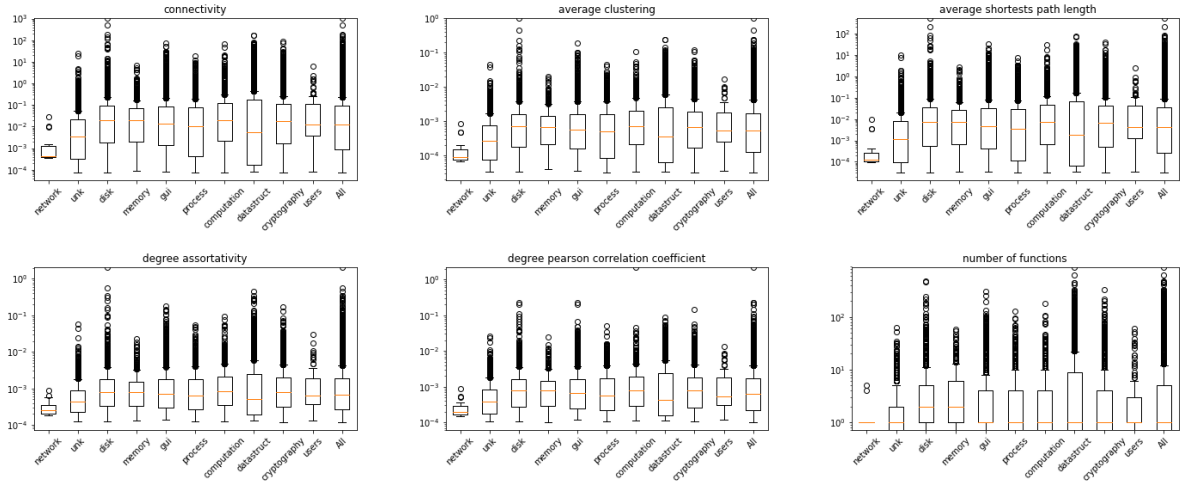

Figure 15: Boxplots of the distribution of the data for several features on v1 dataset

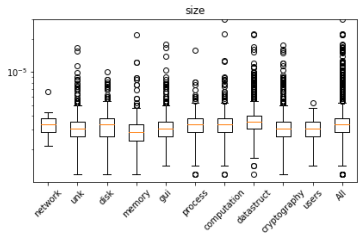

Figure 16: Boxplots of the distribution of the data for several features on v1 dataset

The analysis for dataset v3 is the following:

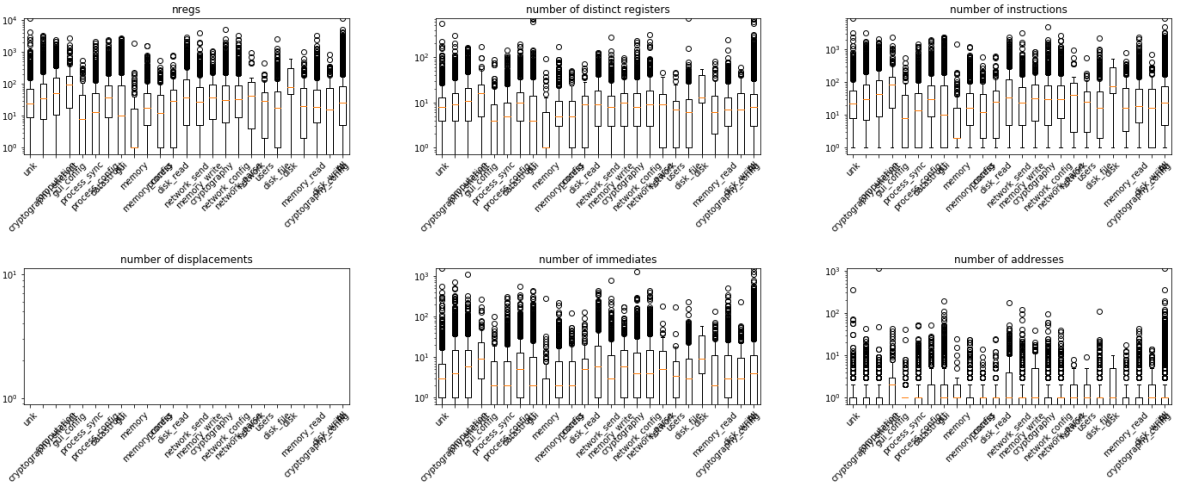

Figure 17: Boxplots of the distribution of the data for several features on v3 dataset

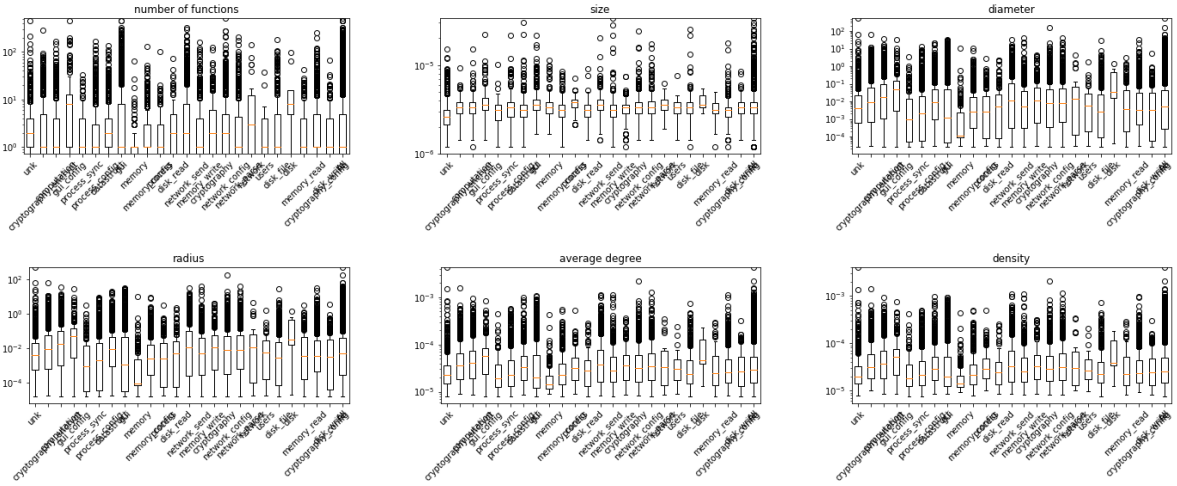

Figure 18: Boxplots of the distribution of the data for several features on v3 dataset

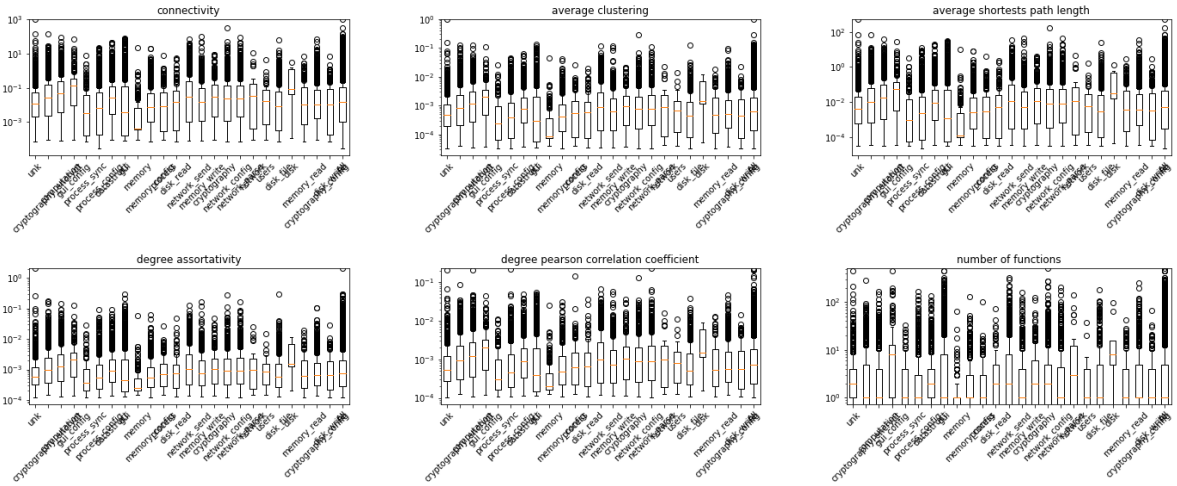

Figure 19: Boxplots of the distribution of the data for several features on v3 dataset

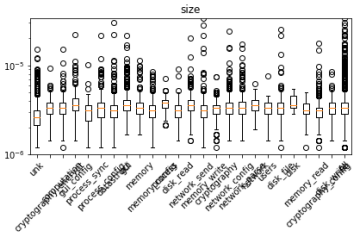

Figure 20: Boxplots of the distribution of the data for several features on v3 dataset

E Training Results

E.1 Experiment 1

Model	Paramteres	RunsEpochs	Splits	Loss	Accuracy	F1-macro
META1	d19d16h10e19n16n15	100-20	100-2000-10556-	2.3561	0.280 ± 0.061	0.251 ± 0.060
META1	d19d16h10e19n16n15	100-250	100-2000-10556-	3.8995	0.139 ± 0.018	0.128 ± 0.020
META1	d19d16h10e19n16n15	100-20	20-500-1500-	3.0269	0.142 ± 0.027	0.127 ± 0.029
META1	d19d16h10e19n16n15	100-250	20-500-1500-	4.9482	0.093 ± 0.013	0.087 ± 0.017
META1	d19d16h10e19n16n15	100-20	200-3000-10556-	2.0677	**0.342 ± 0.078**	**0.312 ± 0.076**
META1	d19d16h10e19n16n15	100-250	200-3000-10556-	3.3414	0.219 ± 0.027	0.208 ± 0.028
META1	d19d16h10e19n16n15	100-600	200-3000-10556-	3.7691	0.142 ± 0.037	0.124 ± 0.039
META1	d19d16h10e19n16n15	100-2	20-500-10556-	3.0165	0.112 ± 0.047	0.088 ± 0.052

Table 6: Edge betweenness approximation with a graph neural network experiment using 28 discrete ranges

The best model is the one trained for 20 epochs, with a dataset split of 200, 3000 and 10556 train, validation and testing number of samples respectively. It attains a 34.3% of accuracy as well as a 31.2% f1-score macro averaged.

E.2 Experiment 2

Model	Features	CV F1-macro	Precision macro average	Recall macro average	F1-macro
MLP2	Code Bow and topological	0.572	0.350	0.291	**0.299**
MLP4	Code Bow and topological	0.528	0.252	0.251	0.237
MLP2	Topological and code attr.	0.420	0.232	0.203	0.186
XGBoost	Code Bow and topological	0.387	0.470	0.374	**0.401**
Random Forest	Code Bow and topological	0.307	0.297	0.293	**0.290**
Random Forest	Topological and code attr.	0.186	0.246	0.198	0.186
XGBoost	Topological	0.171	0.203	0.190	0.190
Random Forest	Topological	0.164	0.199	0.187	0.186
GIN	Graph	0.147	0.122	0.180	0.145
GGNN1	Graph	0.134	0.052	0.098	0.065

Table 7: Function classification experiment results with dataset v1

The best models that have been trained in with the v1 dataset are, in order of appearance in the table:

- and a Multi-layer perceptron with 3 fully connected layers with 94, 20 and 15 hidden units, trained on the bag-of-words embedding and the topological features
- and a Multi-layer perceptron with 4 fully connected layers with 94, 10, 10 and 5 hidden units, trained on the bag-of-words embedding and the topological features
- and a Multi-layer perceptron with 5 fully connected layers with 94, 10, 10, 5 and 5 hidden units, trained on the bag-of-words embedding and the topological features
- and a Multi-layer perceptron with 3 fully connected layers with 17, 20 and 15 hidden units, trained on the bag-of-words embedding and the topological features
- and a Multi-layer perceptron with 5 fully connected layers with 94, 10, 10, 5 and 5 hidden units, trained on the topological features and code attributes
- an XGBoost with a gbtree booster and a maximum depth of 150, trained on the bag-of-words embedding and the topological features
- a Random forest with 4 estimators and a maximum depth of 150, trained on the bag-of-words embedding and the topological features

- a Random forest with 4 estimators and a maximum depth of 8, trained on the topological features and code attributes
- an XGBoost with a gbtree booster and a maximum depth of 150, trained on the topological features only
- a Random forest with 16 estimators and a maximum depth of 150, trained on the topological features only
- a Graph Isomorphism Network with 4 embedding layers, 128 hidden units, a batch size of 128, a learning rate of 0.005, a weight decay of 5e-4 and 100 epochs
- a Gated Graph Neural Network with 1 embedding layer with mean aggregator, 2 fully connected layers of 30 and 20 hidden units, using batch size of 16, a learning rate of 0.001 and a weight decay of 5e-4 and trained during 50 epochs.

Model	Features	CV F1-macro	Precision macro average	Recall macro average	F1-macro
MLP2	Code Bow and topological	0.395	0.224	0.222	**0.213**
MLP4	Code Bow and topological	0.326	0.141	0.187	0.156
MLP3	Code Bow and topological	0.313	0.184	0.176	0.151
RandomForest	Code Bow and topological	0.208	0.288	0.219	**0.209**
XGBoost	Topological and code attr.	0.205	0.236	0.218	**0.221**
GIN	Graph	0.199	0.192	0.205	0.198
Random Forest	Topological and code attr.	0.179	0.196	0.181	0.178
Logistic Regression	Code Bow and topological	0.173	0.223	0.202	0.188
XGBoost	Topological	0.108	0.112	0.116	0.112
RandomForest	Topological	0.102	0.097	0.104	0.098
GGNN1	Graph	0.064	0.106	0.103	0.080

Table 8: Function classification experiment results with dataset v3

The best models that have been trained in with the v3 dataset are:

- and a Multi-layer perceptron with 3 fully connected layers with 96, 20 and 15 hidden units, trained on the bag-of-words embedding and the topological feature
- and a Multi-layer perceptron with 5 fully connected layers with 96, 10,10, 5 and 5 hidden units, trained on the bag-of-words embedding and the topological features
- and a Multi-layer perceptron with 4 fully connected layers with 96, 10, 10 and 5 hidden units, trained on the bag-of-words embedding and the topological features
- a Random forest with 16 estimators and a maximum depth of 8, trained on the bag-of-words embedding and the topological features
- an XGBoost with a gbtree booster and a maximum depth of 150, trained on the topological features and code attributes
- a Graph Isomorphism Network (GIN) with 2 layers and 64 hidden units plus a fully connected layer, trained for 500 epochs only with the graph of each sample as the dataset
- a Random forest with 16 estimators and a maximum depth of 150, trained on the topological features and code attributes
- a Logistic Regression classifier with C of 1, trained with a maximum iterations of 100 in the Newton-Raphson solver algorithm, using multiclass one versus the rest type of classification regions with an L2 regularization
- an XGBoost with a gbtree booster and a maximum depth of 150, trained on the topological features only
- a Random forest with 16 estimators and a maximum depth of 150, trained on the topological features only
- Gated Graph Neural Network with 4 embedding layer with mean aggregator, 4 fully connected layers of 10, 5, 4 and 4 hidden units, using batch size of 16, a learning rate of 0.001 and a weight decay of 5e-4 and trained during 50 epochs.

All models have been trained by searching the best hyper-parameter setup using cross-validation. THe performance metric used is the F1-score with macro average. For neural network models, the loss is used as a

The XGBoost and Random Forest (as well as other Logistic regression and Decision trees models) have been trained with cross-validation with 5 folds, with final retraining, with different sets of features as seen in the results table. The library used is the Scikit-Learn Python library.

The MLP models have been trained with cross-validation and a maximum of 100 epochs, but using the early stopping mechanism to end training when the loss on the validation set starts increasing instead of decreasing, in which case the model is overfitting to the training set. For the MLP training the library used is the Pytorch Python library for deep learning.

The Graph Neural Network models, GIN and GGNN, have been trained with a longer procedure. To perform a successful hyper-parameter search, given the cost in time of epoch of training, has been split into several steps. The first step, has been to train for 5 or 10 epochs with most common models (GCN, GraphSAGE, GIN) with different learning rates, weight decays, number of hidden units, number of embedding layers and batch size. Then, a training with 20-30 epochs has been performed with a bigger set of models (GCN, GraphSAGE, GIN, GGNN, GlobalAttention, Set2Set). Finally, with the selected model type and the selected hyper parameters, a training with different number of epochs has been performed (20, 50, 100, 200, 500 and even 1000 epochs). For each training, there is a cross-validation procedure, where for each fold combination, the selected number of epochs are used to feed the model and back propagate the result to update the weights. After all the fold combinations are trained by the number of epochs, the mean and standard deviation of the performance score is computed on the validation set as well as on the test set. The best model is selected by the best f1-score macro averaged on the different classes on the validation set.

F Code repository

The code and jupyter notebooks used in this project can be found at https://github.com/presmerats/GNN-Master-Thesis .